真正的Windows程序员都应该拥有这本书

Windows 95

用软件组件编写
难以置信的
Windows 95
应用程序

用户控件的程序设计

[美] Paul Cilwa 著
杨开成 罗秀川 译
邢文华 校

人民邮电出版社

责任编辑：周雁飞

Paul Cilwa 是《利用软件组件进行Windows编程》的主要作者。他的畅销书还有《用户控件在Windows程序设计上的威力》和《Borland C++内核奥秘》。除了通过著书帮助广大程序员变得更加多产外，他还是《Visual Developer Magazine (可视开发杂志)》和《Windows Tech Journal(Windows技术期刊)》的专栏作家。

Windows 95 用户控件的程序设计

- ***您是否急于利用新一代的软件组件来改变Windows编程的面貌呢？***
- ***您是否对每次开发Windows应用程序都需从头开始的作法已经厌倦了呢？***

在这本书中，软件组件专家将提供给您利用用户控件来开发Windows应用程序的工具和技术。您将学会如何更好地利用VBX控件、如何植移VBX和OCX以及如何编写出紧凑的、高橹棒性的、真正可重用的组件风格的代码。

Windows 95和Windows 32位编程一般需要使用可以在32位平台下运行的新一代用户控件。但将VBX转换成OCX需要很多技巧和代码。Cilwa提出了一个新的解决方法：建造可以帮助您完成转换的控件，并且这些控件有时会有双重功能。在Cilwa的指导下，您将学习：

- ***编写上乘Windows应用程序所需要的可视化编程技术***
- ***如何建造和使用用户控件代码生成器***
- ***从用户属性到OLE所涉及到的所有OCX编程技术***
- ***如何编写多用途易扩展的控件***

本书是专门针对那些想提高编写16位和32位的VBX和OCX技能的C/C++程序员而设计的。但本书对Visual Basic程序员也会有帮助，因为书中还讲述了如何建造能与Visual C++、Borland C++和Visual Basic配合使用的工具。

ISBN 7-115-06589-6

9 787115 065896 >

ISBN 7-115-06589-6/TP · 473
图字 01-96-1613号
定价：34.00元

Windows 95
用户控件的程序设计

[美] Paul Cilwa 著
杨开成 罗秀川 译
邢文华 校

人 民 邮 电 出 版 社

Windows 95 Programming with Custom Controls

"Original English language edition published by The Coriolis Group, Inc., 7339 E. Acoma Drive, Suite 7, Scottsdale, AZ 85260. Tel: 602-483-0192, Fax: 602-483-0193 Copyright (c) 1996 by Paul Cilwa. All rights reserved."

Windows 95

用户控件的程序设计

◆著 [美]Paul Cilwa
译 杨开诚 罗秀川
校 邢文华
责任编辑 周雁飞

◆人民邮电出版社出版发行 北京崇文区夕照寺街14号
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆开本：787×1092 1/16
印张：23
字数：571 千字 1997年10月第1版
印数：1－6 000 册 1997年10月北京第1次印刷
图字01-96-1613号 ISBN7-115-06589-6/TP·173

定价：34.00元

译 者 序

Windows 程序员都知道，虽然 Windows 标准控件提供给用户许多标准、方便的交互手段，但对程序员来说，编写出漂亮、友好的应用程序界面只依靠那些标准控件是远远不够的。按照传统的做法，您可以按自己的意图将各种标准控件定义成具有定制风格的控件以满足应用程序的需要。但这种一切都从头开发的做法除给程序员带来单调冗长的工作外没有任何价值。长期从事这种开发工作的程序员不会再感受到创作软件的快乐。

现在不同了，软件组件的时代到了！

您可以根据需要先购买或开发那些可用的软件组件，再在“软件建造厂”中创造你的软件。Visual Basic 和 Deiphi 等是“软件建造厂”典型代表。那么当买不到您所需要的组件时，您又如何开发这种组件呢?这就是这本书的教学任务。

VBX 和 OCX 是两种通用的软件组件。这本书的主要内容就是讲解如何编写您自己的组件，而且，您从中还可以学习到如何编写一个代码生成器。本书的 VBX Genie 就是一个很好的例子。此外，您如果想要顺利地阅读完本书，至少应熟悉下面两种程序开发环境：Visual Basic 和 Visual C＋＋。只会使用一种环境的程序员是不适合读这本书的，但愿您不是。

本书共有四个部分。第一部分简单介绍了 VBX 和 OCX 的基本概念；第二部分阐述了如何创建一个“与软件商无关”的 Windows GDI 类库，这个类库是为编写 VBX 服务的；第三部分详细讲解了 VBX 的代码生成器 VBX Genie 的编写；第四部分告诉您如何编写 OCX 组件。其中第一、第二以及第四部分由杨开成译；第三部分由罗秀川译；全书由邢文华校。由于译者水平有限，如有错误不当之处，请读者见谅。

序　言

中国有句古老的祝福："祝你天天快乐"；从某种意义上讲，这句话在我们身上应验了，因为过去的日子很少有现在这样有趣过。作为程序员，我们更是过着有趣的生活。因为我们的工作就是改变这个世界，使得我们的生活越发有趣。而且我们本身也在改变，那些我们用来改变世界的工具也同样在改变，有时它们会以我们难以跟上的速度在变化。

VBX(Visual Basic Extension 的缩写)就是一个这样的例子。VBX 的前身是 Visual Basic，现在已被许多应用程序开发环境和工具所使用，包括 Microsofe Visual C++，Borand C++，Delphi，PowerBuilder 等等。然而，VBX 虽然只有四年的历史，但已经被认为是过时了，正被 OCX(OLE Control Extension)所代替。OCX 标准正在迅速地发展之中。

以前，您可能指望您写的库函数过几年还会有用；现在，如果它们在几个月后仍然有用，您就很幸运了。早在1994年，我在 Coriolis Group(出版商)出版了一本叫做《使用用户自定义控件进行 Windows 程序设计》的书。这本书教 C/C++程序员如何使用 Windows 标准控件在单个动态连接库(DLL)中建造组合的标准控件和 VBX。这本书也介绍了一种有用的 VBX 框架，它允许程序员将注意力集中于自定义控件特定的要求上，而不必为那些每个 VBX 都需要的繁杂工作而浪费时间和精力。

在我写那本书的时候，VBX 的标准，特别是它们被不同编译平台(如 Visual C++，Borland C++等等)所支持的方式还处于不断变化之中。幸运的是，我收到了许多来自世界各地的程序员很有价值的反馈。他们正使用那本书所提供的代码和技术开发他们自己的用户自定义控件(以后简称"用户控件")和应用程序。

随着 OCX 和新开发平台(包括 Visual Basic 4，Visual C++ 4，和32位 Delphi 2)的出现，我认为应该进一步扩展创建软件组件的概念和阐述如何利用新的控件技术来开发 Windows 应用程序。

软件组件——下一代软件

当今您所遇到的大多数应用软件，特别是使用流行的可视(visual)开发环境开发出的软件都可能是在用户控件的帮助下开发出来的。我敢肯定的，您已经意识到了，用户控件正把成千上万的程序员带进了利用用户组件(custom component)开发软件的艺术殿堂。

什么人需要这本书

如果您是这样一类人：您宁愿在业余时间去科罗拉多大峡谷划橡皮筏或在科罗拉多河上用假蝇作饵钓鱼或去喜玛拉雅山脉做长途旅行，也不愿意再编写另一个 Windows 应用程序开发工具，那么，这本书就是为您写的。我的目标是通过教您如何创建自己的可重用软件组件来帮助您节省时间。为了能顺利阅读这本书，您需要具备一些用 C 和 C++编写 Windows 程序

DJS161/11

的经验。本书呈现的用户控件代码是用这两种语言写的。此外，介绍一些 Visual Basic 对您是有帮助的，因为它目前仍然是 VBX 风格软件组件的主要使用者，并且也是第一个使用 OCX 的开发环境。

您将需要什么

为了编译一个 VBX，您必须有一个能生成16位代码的 C/C++编译器。如果您是微软 Visual C++用户，那么您可以用1.52或更早版本的编译器。后期版本不能用，因为2.0版以及更高版本的编译器只能产生32位代码。另一方面，Borland 公司的 C/C++编译器，只要能生成 Windows 代码的都可以。Borland C++4.0或4.2版有一些严重的错误(bugs)，如果您拥用这些版本，您应该要求升级版本。

要运行或测试一个 VBX，您需要 Visual Basic 3.0或 Visual Basic 4.0专业版的16位版本(其它版不含有16位版本)。其它使用 VBX 的平台实际上只能运行1.0版的 VBX。如果您能接受这个限制，您也可以选择它们。

为了在 Microsoft 环境中创建 OCX，您需要安装 Control Wizard。这个软件与 Visual C++同在一张光盘上，但它需要单独的安装。在 Borland C++环境中创建 OCX 需要 Borland C++5.0。

注意，就在我编写这本书的时候，并非所有的平台都使用了 OCX。运行或测试一个 OCX 需要 Visual Basic 4.0或 Visual C++4.0或 Borland C++5.0或32位的 Delphi。

我曾努力使这本书能与尽量多的工具配合使用。但坦率地说，由于两个主要软件商(Borland 和 Microsoft)每经几个月就升级软件——Borland 公司在去年就单独发行了五个版本的 C++环境——我是不可能跟得上的。所以，我没有提供读者如何编译的菜谱式指导，而是力图解释清楚诸如“为什么要设置各种编译开关”等等这方面的问题。因此，您可以根据自己的开发工具来作出明智的决定。

如何使用这本书

本书不是那种包含关于处理所有程序设计问题的资料；但它也不只是一个创建 VBX 或 OCX 的手册(尽管它提供了这方面的信息)。它试图描述1996年组件式软件世界的发展变化，而且不但要帮助您过渡到32位程序代码的编写，还要使您学会处理将来会遇到的这种过渡。

VBX 和 OCX 只是一个借助手段，您要学习的是如何编写紧凑、强壮、真正可重用的组件风格的代码(component-style code)。在这本书中您将要学习面向对象的代码编写技术(可以用 C，也可以用 C++)、如何使 C++重载运算符真正为您所用、如何最大限度地利用新的 Visual Basic 类以及如何设计多用途的 DLL。

如果您只懂一种计算机语言，并且想一直这样下去，这本书就不适合您。而且当今的时代也不适合您。目前，快速开发应用程序的最佳方法是：用 C 或 C++或两者都用来创建工具；再用诸如 Visual Basic 或 Deiphi 这些开发平台作为粘合剂将那些工具捆绑在一起。尽管可以用一个程序员来编写工具；用另一个程序员来使用“粘合剂”，但如果您两者都会做，您的身价就会更高：您的薪水将会告诉您这一点。

本书力图做到读起来很风趣，并且书中的代码也力图做到使您在编写和运行这些代码时

感到很有意思。通常，我会随着代码告诉您一些编程和设计的建议。这些建议是基于我15年来的专业编程和设计经验，它们主要是关于 Windows 的。我的读者和学生普遍认为这些经验是很有用的。

好！现在您可以坐到转椅上，启动计算机，开始阅读吧。如果您在运行 Winsows 95，您应该在启动计算机前花一点时间浏览一下第一章的内容。

本书的内容

本书分为四个主要部分。在第一部分，我们将简要阐述什么是 VBX，如何建造一个 VBX 以及为什么您会需要 VBX。接下来我们以同样的方式来介绍 OCX。在结束第一部分以前，我们简单回顾一下前一本书所谈到的 VBX 框架。

下一部分，我们将开辟一个新的天地：为 VBX 创建 C++类库。为了使它用途更广，这个库必须既不依赖于 Borland 开发环境，也不依赖于 Microsoft 开发环境。我称之为“与软件商无关”的类库。为了创建这样一个类库，我们必须先为 Windows GDI 建一个小的类库。当您做完这部分工作时，您就会感到创建一个有用的类库是多么容易。这时您可以会放下手头的工作，做一个自己的类库玩玩。

第三部分，我们要转换一下所使用的语言。您是不是觉得为重复的编码任务而手工复制代码很讨厌呢？编写一个代码生成器是很容易的，而且也蛮有意思的。用这部分的工程所生成的工具来编写它会更有趣。这个工程就是 VBX Genie，它能自动生成我们前面开发出的代码。

下一部分我们将进一步来学习 OCX。它与 VBX 有什么区别？又有哪些相同之处？在我们学习 OCX 时，我们会先创建两个 OCX。您会发现 OCX 控件非常有用。

走您自己的路

中国还有句俗话：“千里之行始于足下”。我相信您只要一步一个脚印，就一定能学好。所以赶快开始第一章吧，就会很快完成这个特别旅行的。穿上舒适的鞋子！这会很好玩的。

目　录

第一部分　组件式软件的奥秘……………………………………………… 1

第一章　软件组件的威力………………………………………………………… 1

1.1　可重用代码的诞生 ……………………………………………………… 1

1.2　进入 VBX ………………………………………………………………… 2

1.3　OCX 出现了 ……………………………………………………………… 2

1.4　现代的控件怪物 ………………………………………………………… 2

1.5　封装控件 ………………………………………………………………… 3

1.6　我们何去何从?…………………………………………………………… 4

1.7　VBX 还没有消亡的十大原因 …………………………………………… 4

第二章　OCXs：将来的浪潮? …………………………………………………… 12

2.1　巨大的浪潮和内在的不足……………………………………………… 12

2.2　控件怪物………………………………………………………………… 13

2.3　用你自己的眼睛观察吧! ……………………………………………… 14

第三章　VBX 框架的回顾 ……………………………………………………… 15

3.1　VBX 的内部实现 ………………………………………………………… 15

3.1.1　INTERNAL. H 的内容……………………………………………… 15

3.1.2　VISUAL. C 的内容 ………………………………………………… 16

3.1.3　VBXHELP. C 的内容 ……………………………………………… 25

3.1.4　MAIN. RC 文件的内容……………………………………………… 29

3.1.5　MAIN. DEF 文件的内容 …………………………………………… 30

3.2　小结……………………………………………………………………… 31

第二部分　一个 VBX 类库…………………………………………………… 33

第四章　创建一个与软件商无关的 GDI 类库 ………………………………… 33

4.1　C 的缺点 ………………………………………………………………… 33

4.2　C++的优点 ……………………………………………………………… 33

4.3　我们的目标……………………………………………………………… 34

4.4　创建一个与软件商无关的 Windows 类库 …………………………… 34

4.5　GDI 的基本知识………………………………………………………… 35

4.6　您的出发点……………………………………………………………… 36

4.6.1　构造函数和操作符函数的实现 …………………………………… 37

4.6.2　定义 size 类 ………………………………………………………… 38

4.6.3　介绍 rectangle 类 ………………………………………………… 39

4.7　处理彩色 ………………………………………………………………… 41

4.7.1 创建基色类 …… 43
4.8 定义刷子类 …… 49
4.9 不要忘记你的画笔! …… 51
4.10 定义设备环境类 …… 53
4.10.1 创建作图环境(paintcontext)类 …… 57
4.10.2 创建用户区域设备环境(clientcontext)类 …… 58
第五章 创建一个与软件商无关的 text 类 …… 60
5.1 创建 text 类 …… 60
5.1.1 定义头文件 …… 60
5.1.2 text 类构造函数和析构函数 …… 61
5.1.3 text 类的数据存贮 …… 63
5.1.4 text 类的操作符函数 …… 65
5.1.5 谁会更快，是闪电还是超人? …… 67
5.1.6 英雄的助手 …… 68
5.1.7 text 类的加法及加法赋值操作符函数 …… 70
5.1.8 text 类的减法及减法赋值操作符函数 …… 72
5.1.9 text 类的乘法操作符函数 …… 73
5.1.10 text 类的类型强制转换 …… 74
5.1.11 text 类的切分和大小写变换 …… 76
5.2 测试你的新实力 …… 77
第六章 实现 VBX＋＋类库 …… 80
6.1 使用 VBX＋＋：一次预览 …… 80
6.2 VBX＋＋的实现 …… 82
6.2.1 Microsoft Visual C＋＋编译器选项设置 …… 83
6.2.2 Borland C＋＋编译器选项设置 …… 84
6.2.3 创建 model 类 …… 84
6.3 处理方法 …… 106
6.4 control(控件)类 …… 108
6.5 Property(属性)类 …… 111
6.5.1 boolproperty(布尔属性)类 …… 118
6.5.2 colorproperty(颜色属性)类 …… 120
6.5.3 enumproperty(枚举属性)类 …… 120
6.5.4 textproperty(文本属性)类 …… 122
6.5.5 pictureproperty(图片属性)类 …… 124
6.5.6 floatproperty(浮点数属性)类 …… 124
6.6 属性列表 …… 125
6.7 特殊事件和一般事件 …… 126
6.8 去做吧 …… 130
第三部分 VBX 生成器 …… 135
第七章 介绍 VBX Genie …… 135
7.1 代码生成器的设计 …… 135

7.1.1 代码模板 …… 136
7.1.2 代码生成器 …… 136
7.1.3 用户界面 …… 136
7.2 选择一门实现 VBX Genie 的语言 …… 137
第八章 创建用户界面 …… 139
8.1 建立主表 Prime …… 139
8.1.1 编写 Prime 表的代码 …… 140
8.1.2 标签控件(Tab Control)的一些问题 …… 142
8.2 支持 Project 页 …… 143
8.3 支持 Version 页 …… 144
8.4 引入 Listable 类 …… 147
8.4.1 支持集合 …… 148
8.4.2 引入属性函数 …… 150
8.4.3 完成初始化 …… 152
8.5 支持 Model 页 …… 154
8.5.1 Model 页的最终实现 …… 158
8.6 支持 Properties 页 …… 158
8.7 支持用户属性 …… 163
8.8 支持 Events 页 …… 170
8.9 支持用户事件 …… 172
8.10 加入 Methods 页 …… 175
第九章 生成代码 …… 178
9.1 加入代码脚本(Code script) …… 178
9.1.1 创建代码脚本 …… 180
9.1.2 生成代码 …… 180
第十章 自安装 IDE 工具 …… 189
10.1 把 VBX Genie 加入开发环境 …… 189
10.2 给我一个支点 …… 189
10.3 把 VBX Genie 安装到 MSVC 1.5x …… 194
10.3.1 创建全新的 MSVC 工程 …… 196
10.4 把 VBX Genie 安装到 Borland C++ …… 202
10.4.1 创建全新的 Borland C++工程 …… 207
第十一章 为 VBX Genie 创建安装程序 …… 212
11.1 VBX Genie 安装过程记事 …… 212
11.2 是平台，还是沙洲？ …… 212
11.3 Setup Wizard 都做些什么？ …… 213
11.3.1 更深入地探讨安装问题 …… 214
11.4 制订我们的“进攻”计划 …… 216
11.5 创建预安装程序 …… 216
11.6 建造 Generic Genie …… 218
11.7 加入可重用的 About 对话框 …… 221

11.8 创建 Setup Wizard Workaround Installer …… 222
11.8.1 编写可重用的 ProgressBarClass 类的代码 …… 225
11.8.2 展开 Win31文件 …… 227
11.9 更新 SWDEPEND. INI …… 229
11.10 安装 PRESETUP …… 231
11.11 安装 Workaround …… 231
11.12 关于 Setup Wizard Workaround 程序 …… 232
11.13 打点行装回家 …… 241
第十二章 用作查找和替换对话框的 VBX …… 242
12.1 对话框的窘境 …… 242
12.2 Find 和 Replace 对话框 …… 243
12.3 生成 FindReplace 控件的框架 …… 245
12.4 存储和访问 FINDREPLACE 结构 …… 247
12.5 对上下文敏感的联机帮助提供支持 …… 254
12.6 控件的发行 …… 256
12.7 测试 FindReplace 控件 …… 256
第十三章 为标准控件加上工具提示 …… 258
13.1 时代的标志 …… 258
13.2 路标：一直前行 …… 258
13.2.1 选用什么语言作标记 …… 259
13.2.2 为方法编写代码 …… 262
13.2.3 子类方法的原理 …… 264
13.2.4 设置属性 …… 267
13.2.5 VB 中的系统颜色 …… 269
13.2.6 告诫与警示 …… 274
13.3 提示窗口 …… 277
13.4 应用 Tips 控件 …… 281
第四部分 Microsoft 的 OLE Control Wizard …… 283
第十四章 什么是 COM …… 283
14.1 介绍公共对象模型 …… 283
14.2 介绍 IUnknown 接口 …… 284
14.3 OLE 自动化的出现 …… 285
14.4 OLE 控件 …… 286
14.5 内存上的考虑 …… 286
14.6 属性类型 …… 287
14.6.1 库存属性(Stock Property) …… 287
14.6.2 扩展的属性(extended property) …… 287
14.6.3 环境属性(Ambient Property) …… 288
14.7 库存方法 …… 289
14.8 事件 …… 289

14.9 VBX 和 OCX 的比较 …… 290
第十五章 创建一个简单的 OCX …… 293
15.1 创建一个16位的 OCX …… 293
15.1.1 检查和改变控件设置 …… 296
15.2 创建一个32位的 OCX …… 296
15.3 检查代码 …… 299
15.4 支持属性页 …… 304
15.5 支持注册数据库 …… 305
第十六章 使用 OCX 库存属性 …… 309
16.1 设计一个更好的列表框 …… 309
16.2 试运行 TabList …… 310
16.3 了解隐藏的控件 …… 311
16.4 设置默认的属性值 …… 312
16.5 选择一种颜色 …… 315
16.6 使用库存字体属性 …… 318
16.7 建立属性页 …… 319
16.8 继续前进 …… 320
第十七章 用户 OCX 属性 …… 321
17.1 介绍用户属性 …… 321
17.2 从根属性开始 …… 322
17.3 实现枚举属性 …… 325
17.4 瞄准底层控件的属性 …… 330
17.5 种子属性 …… 333
17.6 加入 text 属性 …… 334
17.7 实现数组类属性 …… 335
17.8 你说你想派生一个控件? …… 338
17.9 Appearance(外观)属性 …… 339
第十八章 用户方法和事件 …… 341
18.1 TabList 控件的特殊性 …… 341
18.2 ImageList 控件的特殊性 …… 345
18.3 实现用户事件 …… 350
18.4 结束语 …… 353

第一部分　組件式軟件的奧秘

第一章　软件组件的威力

1869年，John Wesley Powell 领导了第一次科罗拉多大峡谷飘流探险。这是真正的首次探险，因为没有证据表明当地的美国人曾架舟穿过科罗拉多大峡谷(尽管他们肯定在那里生活过)。所以当 Powell 和他的队员踏上旅程时，他们正迈向人们未曾去过的地方。

Powell 是带领九名队员和四艘重木船出发的。Powell 非常肯定在路上会遇到急流，而那些宽底船则是他们通过急流险段的最大希望。尽管如此，飘流结束时，Powell 丧失了两艘船和四名队员。(掉队的四名队员中，有三名至今下落不明)这次探险用了他三个多月的时间。去年，我花了17天顺大峡谷飘流而下，只是为了好玩。同 Powell 一样，我们开进 Lee’ Ferry 渡口，从 Meade 湖离 Powell 出发地不远的地方开跋。路上我们有美味的食物、舒适的睡眠并且有很多时间上岸攀岩或到 Colorado 和 Deer Creek 支流玩跳水。

这是怎么回事呢?

一句话，递增式发展的结果。第一步是 Powell 证明了漂流是可以做到的。

很奇怪，有些事情如果前人已经做过了，而当你再去做它时，会很有安全感的。

下一步是二战期间充气橡皮筏的发展。(这一步也是建立在前面几步的基础上，比如硫化橡胶和充气泵的发明)。

再下一步，也是天才的一步，发生在传奇式人物 Georgie White 把出售战争中剩余橡皮筏的生意和她对大峡谷的热爱结合起来的时候。她首次开创了乘坐橡皮筏飘流科罗拉多河的旅游业。乘客的旅行路线以前已经有人走过了，但是他们是乘坐木制的平底小渔船完成旅行的。这种小船在大的波浪中很容易翻船。而 Georgie 的橡皮筏只会在波浪中吓唬吓唬乘客，乘客们会浑身湿透，感到惊心动魄，但却安然无恙。

1.1　可重用代码的诞生

当70岁的 Georgie White 仍在做着探险旅游业生意的时候，我正忙于在 Tandem 主机(当时我们称它们为‘小玩意’)上工作，使用一种专门语言 TAL 编程。我很快地意识到我可能是在一遍又一遍地编写着许多相同的代码。由于 TAL 是块结构(block-structured)语言(类似 C 语言)，所以我试着将代码放到集中那些完成某一种功能的小函数和过程中。这样这些函数和过程就很容易被重新使用了。我还做了其他程序员很少做的事情：把我的库全部文档化。一年过后，我积累了400多个函数，涉及的范围包括从把二进制值转换到 ASCⅡ 码的小函数到能执行全部功能的“报告书写器”的一组过程。

我经常会使上司和同事感到惊奇，我会在一个下午做完一个月的工作。我能做到这一点是

因为我不再去编太多的程序，只是简单地用我以前写的函数来‘组装’新的程序。四年多过去了，我一直在使用着那个库。

1.2 进入 VBX

可现在有了 Visual Basic。VB 的一部分魅力(也是它成功的最重要的原因)是它的用户控件 VBX。这些控件允许我们程序员扩展程序设计环境。VBX 做到了 Windows 标准控件从未努力去做的事情：建立了生产第三方的、可重用的、可单独发行的程序组件的小作坊式软件工业。软件组件的时代到来了。

第三方 VBX 市场尤如科罗拉多河的十级急流席卷了整个软件开发行业。成百上千的公司投入其中，迅速出现了上千种控件。使用现成控件进行软件开发已经成为了流行的时尚。大量控件等待着程序员们去选择，包括数据库存取、多媒体控制、网络和 Internet 通讯、动画和图形、用户界面工具、文字处理机、文本编辑器、拼写检查器和实时控制器等等，甚至包括专家系统那样的组件。有关 Visual Basic 和其它可视语言的杂志、书籍和学术研讨会迅速涌现出大量文章，纷纷强调用 VBX 建造的 Windows 应用程序更具灵活性、功能更强、可大量节省时间等方面的优点。

但是，没过多久 Microsoft 公司就声称 VBX 已过时了。Microsoft 公司就好象一个发脾气的孩子，拿到了他要的东西，可马上又说那不是他/她最终想要的。尽管 Microsoft 公司自己的产品 Visual Basic 正在使用3.0版的 VBX，但它的另一个程序设计环境 Visual C＋＋却只使用1.0版的 VBX。在我写这本书的时候，Visual C＋＋的32位版本根本就不使用 VBX。

与那些90年代初兴起的软件开发工具和技术相比，VBX 更是担负着对 Windows 软件开发前景的变革。为什么这么说呢?因为 VBX 第一次实现了软件组件的实际应用。这些软件组件可以被嵌入开发平台中，用于建造任何一种应用程序。

Microsoft 公司认为 VBX 标准的缺点是它不能用在32位的环境中。但你要明白，这个缺点是软件厂商人为造成的。没有理由不可以把软件调度层设计成能将32位参数转换成16位参数。毕竟，Microsoft 公司的其它 Windows 代码是这样设计的，允许其它类型的16位 DLL 在诸如 Windows 95和 Windows NT 那样的32位操作系统中正常运行。

1.3 OCX 出现了

Microsoft 公司一个最新的观点是所有的东西都是 OLE(对象连接与嵌入，Object Linking and Embedding)世界的一部分。虽然 VBX 可能要被迫顺应这种观点，但 Microsoft 公司已决定放弃它，而用 OLE 控件来代替。这些 OLE 控件通常被称为 OCX。

在下一章我会多讲一些关于 OCX 的知识，并且本书的后两部分有关于 OCX 的更详细阐述。现在你只需知道 OCX 在应用于软件组件的功能上与 VBX 是等价的，但它需要的系统资源是 VBX 的两倍到十倍，却没有为程序员提供更多的价值。

1.4 现代的控件怪物

作为一个 Windows 用户——您肯定是，对吗?——您一定很熟悉控件。它们就是那些你在

处理数据时所指向、点击和要拖动的东西。从前，这些特征曾属于卡片分类机或打印机上的硬件控件。现在，他们只是屏幕上的一些图，但他们可以对键盘和鼠标做出反应。我们喜欢使用这些控件。

作为一个 Windows 程序员，您可能熟悉控件的内部机制：每个控件都是一个 Windows 窗口，这个窗口有自己的窗口处理函数，可以接收来自其它窗口的消息和向其它窗口发送消息，当用户指向、点击或拖动它时，它会将用户的操作通知给父窗口。作为一个 VBX 程序员，你可能会意识到在标准 Windows 窗口上面还另外有一个层次将控件挂到 Visual Basic、Delphi、PowerBuilder 或 Borland C＋＋开发环境内。这个层次不但支持工具箱(toolbox)中各种控件的图形显示，而且支持以“属性”和“事件”来命名控件的功能以及将对属性和事件的引用转换成向底层控件发送的消息。

OCX 标准也是建立在 Windows 窗口基础上的。尽管在许多方面它重复了 VBX 的功能，但它实现的方式是如此的复杂和深奥，以至于在 Microsoft 公司将那些复杂的东西用 C＋＋类库隐藏起来之前，是没有人能够弄得懂它或让它正常运行。事实上，第一部有关 OLE2.0程序设计的书的作者已经承认虽然他在写书，但他的确弄不太懂。

然而，Micorsoft 类库确实使我们能够编写 OCX。去向神灵祈祷吧，千万别发现那些只有深入 Microsoft 源代码才能确定的错误！

写得好的用户控件有一个优点，就是它把要完成的功能全部封装起来。为了更清楚地说明这一点，我们把用户控件比做电视机。你知道电视机内部充满了元器件、电路板以及发热的和带电的东西。你当然不会愚蠢地打开后盖，在里面焊点什么东西。你也知道你不必那样做。所有与你有关的控制器都被放置在机器的前部、后部或边上，或放在遥控器里。

好的用户控件也应提供给程序员清晰的流水线式外观。程序员可以而且也应该能控制一些东西(如属性、事件和方法)，但那些程序员并不需要接触的东西都应隐藏在控件内部，不能让程序员见到。

使用这个模型，你可以通过用户控件传递任何有用的东西。控件并不一定非是终端用户使用或看到的，有时它只是为了方便程序员而已。比如，Visual Basic 的计时器控件在运行时是不可见的，它只是为了方便程序员而简单地将计时器函数封装起来。

1.5 封装控件

你来封装一个标准 Windows 控件吧，要确保 Windows 消息是与它进行通讯的唯一方式。老兄，那里可没有全程变量！

当在标准 Windows 控件基础上建造一个 VBX 或 OCX 时，你要为控件定义你自己的属性(对 VBX 和 OCX 来说)和方法(只对 OCX 来说)；当那些属性和方法实际运行时，你要把适当的 Windows 消息传递给底层的 Windows 窗口。你还要定义一些事件(event)，当底层窗口向 VBX 或 OCX 层(底层窗口的父窗口)发送通知消息时这些事件会被激活。

我想，标准 Windwos 窗口类型的用户控件一直没有过时的主要原因，其一是以前 Windows 程序是用 C 写的；其二是 C 程序员都有从不使用别人代码的怪癖。他们不喜欢封装的控件；他们更喜欢不带机箱、显像管没有固定在底座上的电视机。但当他们发现他们已被 Visual Basic 程序员淘汰时，他们就放弃了原来的习惯。或者说，旧风格 C 程序员在达尔文适者生存的竞争中就这么灭绝了。当今的程序员必须精通好几种语言(恐怕很少有人不是这样)，并且能使

用多种工具来开发应用软件。他或她首先尽可能地寻找各种可用的组件，当找不到适用的组件时，又能创建新的组件。并且只有他或她找不到合适的组件时，才去开发新的组件。

1.6 我们何去何从？

既然这样，你要做的是什么呢？是编写据说只有几天活头儿的VBX呢？还是编写目前只有屈指可数的几种环境中才能使用的OCX呢？还是放弃编程，去大峡谷当一名导游？如果我必须选择，我选择去当导游。然而，这里还有一种选择，第四个选择：VBX+OCX。

完全可能编写出双倍OCX功能的VBX。

只有你真正理解了VBX和OCX的编程要求并且你擅长编写结构化风格的程序，第四个选择才是可行的。这种结构化的编程风格要求程序员能编写出小型的、高聚焦(一个函数只完成一种功能)的函数。接下来让我们来看看为什么VBX还会生存下去(这使得我们值得去做第四个选择)和为什么VBX给OCX编程提供了一个坚实的基础。

1.7 VBX还没有消亡的十大原因

除了写书和划船，我还从事Visual Basic，VisualC++和BorlandC++的教学工作。因为这三个编程环境都声称支持VBX，所以班上的学生经常问“我一定要学习编写VBX吗？”、“VBX还没过时吗？”

之所以有谣言说VBX已经过时或即将过时了，主要是因为VBX是按16位代码标准编写的，而现行的Windows版本(比如Windows NT)是按32位标准运行的。由于VBX运行在16位代码段中，人们担心它不能适应新的环境。Microsoft公司不想把VBX标准扩展到32位而是用另一种新的标准OCX来代替它的做法更加重了人们的这种担心。

所以，开发者到底应怎样做？是继续编写随时都会过时的工具呢？还是再次抛弃以前编写和购买的工具呢？我们到底要这样做多少次呢？看起来好象无论什么时候，只要我们准备贮备可重用代码，Microsoft公司总是给我们来个釜底抽薪。所以，我们程序员总是生活在无措手足的状态中就不足为怪了。

但别灰心，事情并不象看起来那样糟糕。尽管你很快就会想学习编写OCX(可能你在读完这本书后就会这样做)，但这并不意味你在开发或使用VBX过程中所获得的技巧就毫无用处了。这里有一些理由足以说明“VBX已消亡”的观点是被人夸大了。

#1：现在你的客户需要应用软件

我们很容易忽略这样一件小事情：按最初的计划，Windows 95应至少提前一年发行。正如你看到的，当Microsoft公司放弃使用所有原来的工具时，他们也遇到麻烦。可你的客户却不打算扔掉他们买的任何东西——至少现在不想立即这样做。一些客户仍在使用Windows 3.1。使用Visual Basic 3.0来开发软件将会持续一段时间。Microsoft公司也清楚这一点，所以它的软件都设计成是向后兼容的，以允许这种情况的存在。

但现在你的客户是有需求的。并非我们每一个人都能等到神话的降临：我们用来开发应用程序的平台达到了尽善尽美、登峰造极、不需要再更新版本的程度。你想想看，这一天是不可能到来的。推出新版本软件的源动力不是对尽善尽美的追求，而是更高利润的追求。只要用户

愿意升级他们的操作系统，操作系统软件商就愿意提供升级。事情就这么简单。那么，你用什么工具来快速开发应用程序呢?还跟以前的一样用 Visual Basic、Visual C++、Borland C++、Borland Delphi?软件组件会帮助你进一步加快开发速度。你一定会想到它：VBX。

现在，如果你要编写 OCX，你可以把它嵌入到 VB4.0的应用软件中；你也可以在 VisualC++4.0和32位 Delphi 应用中使用它。但你现在知道如何编写一个 OCX 吗?不知道?我不这么看。是的，你最终是要掌握如何编写 OCX，并且这本书也会帮助你达到这个目标。尽管如此，VBX 仍会给你一些帮助。你在编写 VBX 过程中所付出的努力是不会白费的，因为：

#2：**Windows 95 支持现存的16位代码**

记住我先前所说的：只要人们愿意买，就会有新版本的操作系统被推出。如果新版本的操作系统迫使用户不得不放弃原有的应用软件，用户就不会买它。那些应用软件所含的价值太高了，不可能一下子被替换掉。(实际上，我的客户已经开始抱怨他们不得不频繁地更替应用软件了。比如，他们认为如果 Microsoft Office 新版本增加了工具条，但这些工具条并不意味系统功能的改进的话，花钱去升级这个软件是很不值得的。)所以，就象 Windows3.1完全支持 Windows3.0的应用程序一样，Windows 95不得不支持 Windosw 3.1的应用程序。那些应用程序有很多是用 Visual Basic 或 C 写的，并且简单地重新编译并不能将它们升级到32位。既然如此，我们还能做些什么呢?

要想明白这种局面是怎样形成的，你必须知道一点有关 IBM 兼容机的心脏部件 Intel 80x86芯片的知识。(PowerPC 芯片当运行与 Intel 兼容的操作系统时会模拟 Intel 芯片，所以我在这里所讲的也同样适用于 PowerPC)

你可能知道，DOS 程序都是在实模式(real mode)下运行的。“实”是指程序中的寻址是真正的存取地址。也就是说，地址0000：0000就是 RAM 的起始地址。地址0000：0001是下一个字节，等等以此类推(见图1.1)。由于实模式被设计成在某一时间只能运行一个程序，因此没有必要提供防止向另一个程序的内存空间进行意外写入的保护。当指针被破坏或程序员忘记初始化指针时通常会出现这种意外写入的情况。

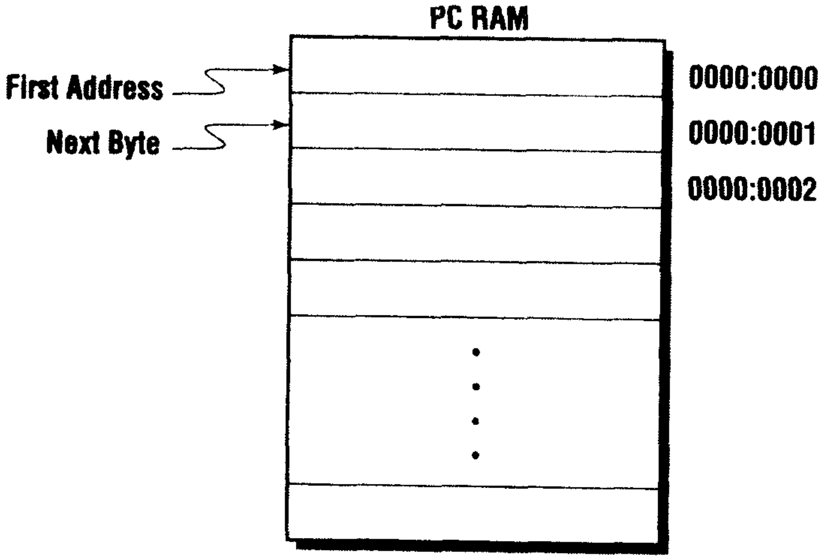

图1.1 在实模式下存取 PC 机内存

在保护模式下存在一个保护层。应用程序申请的每一块内存都由一个描述符来定义。描述符的索引被保存在32位的地址中最重要的13个位上。如图1.2所示，程序只能间接地使用内存。你不用担心，一切都会顺利运行的，因为 CPU 自己能以芯片层的速度自动管理这些描述符。

描述符提供了各种有用的信息，如这块内存指向的是数据还是代码，它是否可以写入以及它的大小等等。

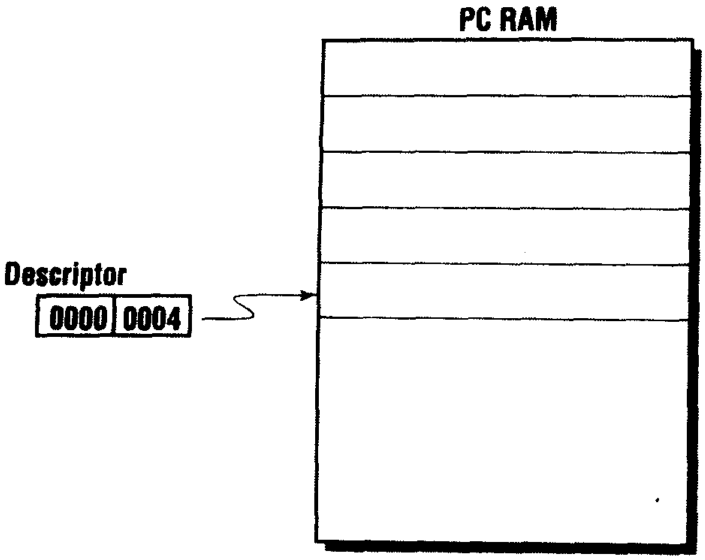

图1.2 在保护模式下通过描述符存取内存

CPU 运行在保护模式下时，至少要维护一个描述符表。这个表叫做全局描述符表(global descriptor table)或 GDT。但 Windows 并没过多地使用 GDT。而是让每一个运行的任务使用一个局部描述符表(local descriptor table，LDT)。不同任务的 LDT 可以有完全独立的一组描述符，指向各个内存块。如图1.3所示，在保护模式下是通过这种机制来提供“保护”的。

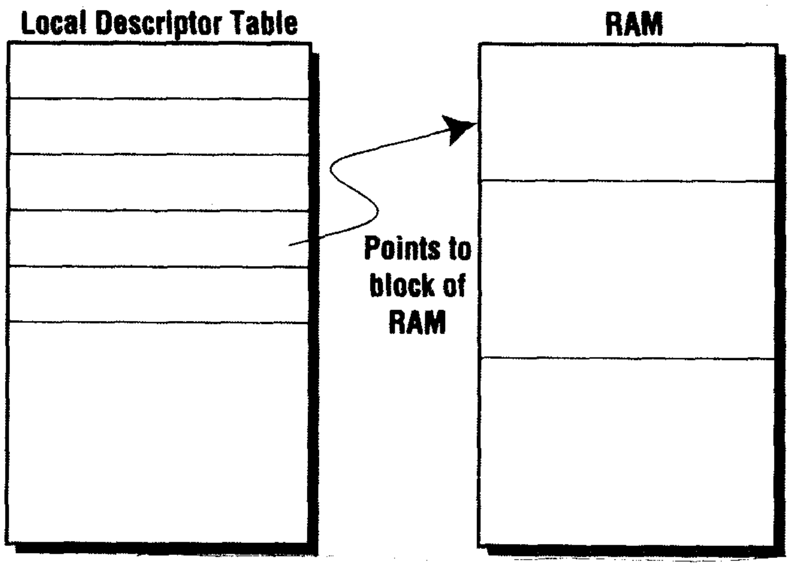

图1.3 保护模式下通过描述符表建立保护

但是，Windows3.1并没有充分利用这种机制。每一个 DOS 程序作为单独运行任务都有自己的 LDT，而所有的 Windows 应用程序却同时共享同一个 LDT。这就是为什么一个崩溃的 Windows 应用程序就会使整个系统变得不稳定的原因。当用户遇到这个问题时，我们的回答就是“试着重新启动 Windows，看看这个问题是不是不见了。”

在 Windows 95或 NT 中，32位应用程序并不共享 LDT。这是 Windows 比以前更加稳定的重要原因之一。但是，为了支持现有的16位应用程序，Microsoft 公司将这种理想的机制做了变通。所有16位应用程序都运行在一个单独的16位任务中，这个任务中所有程序都使用同一个 LDT。所以 Windows 95中的16位应用程序并不比 Windows3.1下稳定多少。如果一个16位应用程序崩溃或没有很好地中止就仍有可能使16位任务变得不稳定。32位的应用程序没有这种隐

息。但是恢复16位任务的稳定性的唯一办法——正如你所猜到的——还是重新启动Windows。

还记得我提到的LDT中的描述符吗?我说它们包含着许多有用的信息，包括所描述的段是代码还是数据等。如果该段是代码段，该描述符也描述了它是16位的还是32位的代码段。

实际上，存在问题的唯一原因是当Windows转到32位时，几乎所有的标准参数都要从16位变成32位的。现在Windows系统中窗口句柄(handle)是32位的，而不是以前的16位的了。原子(atom)也是32位的，刷子句柄、画笔句柄、设备上下文句柄等等都是32位的。甚至每个窗口过程的wParam参数也被扩展成32位的了。所以只使用CPU来处理16位代码是不够的。实际上每一个过程的所有的参数都必须自己完成这种转换。

OLE2.0标准要求既支持16位代码，又支持32位代码。参数的转换是通过一个简单接口完成的，而且应用程序的性能并没有降低得太多。(但软件性能的降低足以促使用户尽快升级到更新更快的32位版本。注意，对于那些认识到这种升级是一种性能的提高的人们来说，Windows 95下的新版软件不必真的要比在Windows 3.1下的老版本快)这种转换机制叫做调度(marshaling)。

所以，最后的结论是：虽然VBX不直接面向32位环境，但是32位环境仍然支持使用VBX的应用程序。要知道升级到Windows 95并不需要立即升级你的应用软件，所以如果需要，你仍可以继续编写和使用VBX。

你可能最终将你的控件升级到32位的OCX(这是这本书后面要讲的)，可这种升级有没有可能会带来什么不好的后果呢?而另一种做法则是直接编写OCX，在有限的几个支持OCX的开发环境中开发客户应用软件。与OCX标准相比，

#3：VBX具有定义得很好的标准

VBX标准之所以很容易使用，一部分是因为(相对来说)它已经存在很长时间了。事实上，尽管自从Visual Basic 1.0开始增加了许多内部的属性、事件和方法，但它本身的结构并没改变。之所以这样幸运是因为只有1.0版本的VBX才能被一些版本的VisualC++或BorlandC++应用程序所使用(实际上，这个限制存在于MFC(Microsoft Foundation Classes，微软基类库)和ObjectWindows中，而不是编译器中)。

与此相比，OCX的所谓标准还不能真正称为一个标准。当然，Control Wizard(以及其它工具)可以产生框架OCX(skeleion OCX)，而且后面我们将使用这个工具产生框架OCX。但Control Wizard的任务只是从那些使用OLE2.0的基类中派生出新的C++类，而且这些类并没有被很好的文档化。Microsoft公司经常根据他们的设想任意改变OLE2.0的内部结构．在他们从中真正获得一些性能的改进之前，有谁知道他们所做的改变会不会对你的控件造成负影响呢?尽管OLE2.0的支撑结构不会改变太多，但确实会有一些改变的。你只有学会现在的OCX，才有可能处理OCX以后的变化。对于现在的应用程序来说，你仍需要编写VBX，幸好……

#4：VBX是很容易设计和编写的

虽然VBX看起来与标准Windows子控件很不一样，但实际上差别并非那么大，而且差别被定义得很清楚。最明显的差别是VBX的程序员接口被建在VBX自身的内部。也就是说，属性和事件的名字——字符串本身——在VBX中经常使用的是固定字符集。首先这破坏了Windows语言的国际化。但你很快就会发现这只影响到了工具的语言国际化，而并没有影响到其它方面，应用软件本身可以在法国或上沃尔特或其它什么地方发行，因为VBX中的属性

名对终端用户是不可见的。

标准本身是在控件开发工具箱(Control Development Kit，CDK)中定义的。这个工具箱包含在 Visual Basic 3.0专业版中。(Visual Basic 4.0中并不包含 CDK)在相应的文档中每个标准属性和事件都标有版本号，所以如果你想编写1.0版的 VBX，也是能做得到的。当然，所有真正 COOL 的属性都加到1.0以后的版本中了。但你并非无事可做噢！

VBX 风格的控件通常是在底层标准窗口的基础上实现的。事实上，只有3.0的 VBX 才能创建一种特殊类型的控件(叫做图形控件(graphical control))，这种控件并没有底层窗口。VBX 控件的结构是在底层窗口基础上构建的。

在下一章我们将检验一个适于派生你自己的控件的用户控件框架。这个框架显示了 VBX 的基本格式。首先，初始化的结构定义了每个用户属性。而后，说明一个数组，并将它初始化为每个标准属性和用户属性。标准事件(event)和用户事件也使用这种模式。最后，象窗口过程那样，要实现一个消息开关(message switch)。但这个消息开关是被控件过程(control procedure)调用的，并且它带有另一个参数：控件的句柄。

就像最初 Windows 所采用的将代码封装在一个窗口中的方法一样，整个这种机制允许程序员使用面向对象程序设计技术，既使使用的不是面向对象语言(如 C)。

当你想编写一个不同于任何一个标准窗口类的 VBX 时，即使你不显式说明，你也能以默认的方式获得一个。你获得的是一个一般的窗口，它除了允许 VBX 消息重定向机制运行外，什么都不做。但……

#5：VBX 可以在现存窗口的基础上编写

你一定不只一次地想用 VBX 重新实现一些你或其他人以前所写的标准窗口类。你甚至还记得对标准 Windows 类进行的改进，如类似 VBX 的用户自画列表框控件或只接收数字输入的编辑控件。你当然正迈向 VBX。你不必关心 VBX 的底层窗口是如何工作的。你只需知道它的 API(应用程序接口)，就象你在标准 Windows 应用中使用控件一样。你最大的任务是把控件的 Windows API 转换成 VB 程序员都熟知的 VB 接口。你随时都可以借用那些“标准”属性和事件的命名方法。比如，如果你新的 VBX 要包含一种事物的列表，你可以定义 List(列表)数组和 ListCount 两种属性。列表框 VBX(Listbox VBX)和组合框 VBX(Combobox VBX)就支持 List 和 ListCount 这两种属性。当你需要扩充预定义属性名时(你可能需要定义类型完全不同的两种列表)，你仍要尽可能地按照这个模式去做，例如 NameList、NameLisCount 和 CityList、CityList Count。同时，你要注意为每种属性所设置的标志。你对属性可以有许多控制：属性是只读还是只写的?在设计时还是在运行时还是在两种情况下都可以存取?在底层窗口生成以前就可以使用呢还是在底层窗口生成之后才可以使用等等。你要明智地选择这些控制。

一句话：要把所设计的东西看作黑匣子。标准 Windows 控件是黑匣子；我们所设计的又是包含这些黑匣子的新黑匣子。

如果你的方法正确，你就会设计出非常易用的控件。程序员不需要过多地(或根本就不需要)参照相关文档就可以使用它们。用这种方式来设计工具意味着你必须学习面向对象的编程技术，即使你现在还不熟悉 C++，因为……

#6：VBX 能教给你面向对象的程序设计方法

我不管你多么不情愿，现在你只有两个选择：要么学习面向对象的程序设计，要么换一

个新的职业，你一定觉得面向对象程序设计技术很难学，特别是有人站在计算机科学的高度上向你解释面向对象时，你会觉得面向对象程序设计技术深不可测。幸运的是，面向对象的程序设计技术比你想象的要容易学得多。面向对象程序设计最重要的前提是我们人类天生就能理解“对象(object)”。面向对象程序设计技术只是简单地将我们已经掌握的处理事物的能力应用于解决计算机问题。并且，老兄，你不必使用“面向对象语言”(如C++)就可以进行面向对象的程序设计。

OOP(面向对象程序设计)的最重要功能之一就是使你将注意力集中于你当前所编写的程序段。下面是一个C程序员熟悉的例子：C标准函数库中的一组文件操作函数。这些函数使用了数据类型FILE，如

```
FILE*MyFile=fopen("MYFILE.DAT","w");
fwrite("this is some text",1,25,MyFile);
fclose(MyFile);
```

正如你所知道的，如果你去查找FILE是如何定义的，你会费力不讨好。MyFile只是一个指向一个结构的指针。至于这个结构到底是如何定义的?谁管它!这就是关键所在。你无需知道fopen()、fwrite()和fclose()的内部操作，它们会正确运行的。这就是面向对象程序设计(至少在封装方面是)。你可以做同样的事情。把你的程序设计任务分成许多小的部分，对每一部分的程序进行编码，并且用“控制块”或象FILE那样的结构将这段程序所需要的数据隐藏起来。用面向对象的术语来说，这个结构包含着你的“属性”；对那些属性进行操作的函数就叫做“方法(method)”。

事实上我更喜欢用C而不是C++来编写VBX，因为如果你想在一个不是使用你所选择的类库所建造的环境(比如说Visual Basic)中使用你的VBX时，你很难在你的VBX中包含MFC类或ObjectWindows类。并且如果没有一个好的类库作支持，C++并没有什么比C更好的地方。但是我为VBX编写了一个C++类库，它对我们最终学习编写OCX会有所帮助的，因为OCX开发工具箱包含了许多使用方便的类来简化编码工作。尽管如此，C可能是你应选择的语言。你要谨记，你可以用非面向对象的语言来编写面向对象的代码，并且这样做会有双倍的好处，因为……

#7：VBX向Visuai Basic 3.0中加入了对象

虽然可以用上面提到的技术在Visual Basic中编写面向对象代码，但那样做可能是令人讨厌的，因为Visual Basic 3.0中已经有“对象”了(这个版本只是不希望程序员建造自己的对象而已)。Visual Basic 4.0则鼓励程序员建造自己的对象，但如果你因为某种原因不能升级到4.0，并且你是一个懂得C的VB程序员，那么，编写VBX是一种向Visual Basic 3.0环境中加入对象的方式。

你还记得我说过的“VBX不一定非是可见的控件”吗?计时器控件和CmDiaiog控件都是非常好的例子。在传统的意义上讲，它们根本就不是控件。终端用户根本就看不到或无法(哪怕是间接地)操作它们。但是我们要知道，Visual Basic中的控件就是带有属性、事件和方法的对象，因此，它们确实就是控件。你也可以这样做。比如，假设你想封装一个逻辑函数(比如邮政编码的寻找算法)，使程序员不用关心这个函数的内部机制，你就可以通过编写一个运行时不可见的VBX来封装它。这样，如果为一个有效的邮政编码设置一个ZipCode(邮编)属性值，就可以读取City(市)和State(州)这两个属性值，进而返回邮政编码所在地的地址。反过来也一

件，设置City(市)、State(州)、Address(地址)就会直接从属性中正确读取ZipCode。程序员既不会感到迷盲也不会感到大惊小怪，更不必去参照联机帮助信息弄清控件是如何工作的。当然，函数的内部发生了许多事情：如数据库的查找等等。但是控件使用起来很简单，这是最重要的。复杂性都是内部的、隐藏的。

此外，即使当不包含数据时…

#8：VBX为无数据对象提供了很好的控制点

Comm控件是另一个不可见控件。它不包含任何数据，尽管它可以做为传输数据的通道。它所做的(而且也做得很好)是将那些在PC机上进行I/O(输入输出)操作所需要的东西封装起来。有些事情它已经自动为你做好了，而有些事情则需要你去显式说明；但说明格式被规范得非常简洁并有详细的文档。从各方面来看，Comm控件(不是Microsoft公司提供的)都可以称为一个相当好的使用VBX来封装一个设备或一组相关函数的例子。

既然这样，CmDialog控件可以为VB程序员提供一种访问公共对话框动态连接库(Common Dialog DLL)的手段吗?当然可以，自从Visual Basic允许使用DLL中的函数起，它就已经有这个能力了。尽管这种访问一点也不象Visual Basic风格，但CmDialog控件非常受欢迎。

但是CmDialog有一个小小的失误，就是它不能访问查找和替换对话框(Find and Replace diajog，它也存在于Commom Dialog DLL中)。因此，编写一个以VB友好的风格来包装那些控件的VBX则是个好主意。(但不要今天晚上就开始编写，除非你急需这个控件；这样的控件是本书后续章节要阐述的主题)

总的来说，为Visual Basic创建这些对象是对Windows的扩展。因为按最初的设想，从面向对象的意义上讲每个Windows窗口都是一个对象。这就是它为什么这么COOL以至于……

#9：VBX可以按三种标准编写

一般地VBX要被编写成是VBX而不是其它的东西，但没理由非得这样。当然，最初的VBX控件(存在于VBRUN300.DLL文件中)都是基于标准Windows控件的，并且除了VB界面外什么都没增加。并且随Visual Basic发行的VBX都只增强了Visual Basic(以及能使用VB的其它环境，如MFC和ObjectWindows)的功能，并没增强“标准”Windows应用程序的功能。但这并不意味你必须把自己局限于此。

记住，VBX可以建造在任何底层“标准”窗口的基础上，包括你随便写的标准窗口。这意味着你可以在一个DLL中(VBX就是DLL)包含你的代码，甚至可以包含Borland公司的资源编辑器(Resource Workshop)的标准控件挂钩函数，然后在此基础上加上可视程序员接口和VBX挂钩函数。

此外，如果你能编写小型的高聚焦的函数，你可以将VBX控制代码加入到OLE Control Wizard所生成的OCX中，这样所有三种环境(Visual Basic，MFC，ObjectWindows)都可以使用它了。

关键是你要坚持始终使用面向对象的思想和方法来考虑问题以及进行软件设计。当你设计一个“标准”控件时，要从Windows消息角度来考虑如何设置和读取属性(数据)和激活操作(方法)。控件向它的父窗口发送的WMV COMMAND消息就相当于“事件”。你要为C和C++程序员提供一个可以包含在(#include语句)他们的程序文件中的头文件。

Borland公司的资源编辑器的挂钩函数是可选的。如果你包含它们，你的应用软件程序员

就可以将那些新控件插入到资源编辑器的环境中，就好象它属于那个环境一样。如果你的程序员有的仍在使用 Microsoft 公司的对话框编辑器(在旧的 SDK 中)，这些控件也同样能正常工作。但很不幸的是，Microsoft 决定放弃他们所建立的标准。App Studio 并没提供对标准风格的用户控件的访问。

当你在标准层上加上一个可视层时，在填充模型结构时你只需简单地说明新的底层窗口类就可以了。然后设计用于映射消息的属性和事件以及底层窗口类的 WM—COMMAND 通知消息。如果你有一个在底层窗口中实现的"方法"，但没有相应的名称(你可以为属性和事件起名字，但不能为方法起名字)，你可以使用 CmDialog 和 Ole2.0控件所采用的方法：定义一个 Action(操作)属性。给它设置一个数字(请为程序员提供一个常量)并且将这个数字转换成发送给底层窗口的消息。真是小菜一碟！

接下来，当你最后准备好时…

#10：OCX 可以很容易地从 VBX 派生出来

是的，OCX 标准可能仍是在 C++类库中隐藏着的被严密看管的机密。不过，你大可不必理睬这些。OCX 在形式上与 VBX 非常相似。OCX 也有属性、事件和方法。最好的一点是，从开发者的立场上看，OCX 允许用户自己为方法起名字。VBX 开发工具箱的最讨厌的一点就是不能给方法起名字。

毕竟，特别是当你基于用户自定义的底层窗口来编写 VBX 并且那些底层窗口的可视部分只是窗口界面时，将 VBX 转变成 OCX 就应该是非常简单的。当然这就意味着你就不必担心你编写的工具是否随时都可能过时了。

第二章　OCXs：将来的浪潮?

以前，Microsoft 为我们展示了一个令人兴奋的新前景：一种新技术，一种建构在 Microsoft Windows 基础上的东西，一种可以使应用程序相互交谈、相互控制和共享数据的东西。这门技术有一天会使在一台计算机上运行的应用软件能与在另一台计算机上运行的软件共享数据并且能控制另一台机器上的软件运行。按照设计者的意图，这门技术将会成为将来的浪潮。我们这些程序员也被鼓励在我们的软件中使用它，但我们是在冒着这些软件很可能在被用户购买前就被淘汰的风险。现在，事情变得不那么容易了。这门新技术是建立在没有很好定义的规范的基础上的，而这些规范又很难被程序员们所理解。但我们曾试图接受它和使用它，并且非常肯定的是许多软件已经使用这门技术了。

我当然是指 DDE(Dynamic Data Exchange：动态数据交换)。现在，如果你认为我谈论的是 OLE(最初的 OLE--我们现在叫它 OLE1.0)那也是可以理解的。毕竟，Microsoft 公司是使用和 DDE 完全一样的手段来宣传和出售 OLE 1.0的。而且如果你认为 OLE 1.0除了名称外还有其它方面与 OLE2.0相同，你一定还没有学习相应的 API(应用程序接口)。我认为这门新技术之所以最终叫做 OLE 的唯一原因就是 Microsoft 公司想掩盖他们又一次给我们程序员来了个釜底抽薪的事实。(实际上，它的真正的名称叫做 COM，即“公共对象模型”)

这难道意味着你不应该去开发使用 OLE 的应用软件了吗?根本不是!这是一项很重要的技术。它试图要解决的“使计算机程序能相互交谈”的问题更是具有重要意义。但是，除非你乐于做此类事情，我是不会建议你花时间去费力研究 OLE 2.0的实现机制的。MFC(Microsoft Foundation Classes)提供了一种有用的 OLE 2.0接口。由第三方提供的其它类库也使程序员可以使用这项技术。但它的内部机制太复杂了，所以对一般程序员来说，花时间去弄懂它是不值的。就象 DDE，在 DDE 管理库出现之前，没有人愿意研究和使用它。鉴于过去的经验，到底会不会很快出现替代 OLE 2.0的“OLE 3.0”呢?

2.1　巨大的浪潮和内在的不足

OLE2.0是基于 COM 原则设计的。COM 是公共对象模型(Common Object Model)的缩写。其主要思想是以与语言、机器、甚至平台无关的方式来表征对象。COM 试图成为双向标准并且有人认为可能用任何一种计算机语言都可以实现 COM 对象。那么，最终结果呢?正是如此，可重用软件组件就是一个例子。这些组件在复杂性和功能上所涉及的范围很广，从简单的按钮到完整的字处理器，无所不包。

VBX 是可重用的，因为它的属性、事件以及方法都是公开的。也就是说，包含它的应用软件通过询问就可以知道他们是干什么的。COM 对象的核心被认为要比 VBX 简单并且支持与机器和语言的无关性。所以它公开了它的接口，即包含它的应用软件可以用来操作它的一组函数。

接口就是一组函数头，包括参数类型，但不含有实现代码。显然，在某个地方必须有它们的实现代码。但按照真正的面向对象的习惯，那些内部实现并不是使用 COM 对象的人所关心的。那些函数应是与语言无关的，但实际上，C++是唯一一种非常适合描述它的语言，因为

所有的函数都是C++所称谓的虚函数。

按照最初的设计，所提供的接口要比现在少一些，并且每一个接口都要公开，而每一个COM对象都可以使用对它有用的接口。为了使应用软件知道从哪里开始，有一个接口必须要求所有的COM对象都要实现。它就是IUnknown。从这个接口出发，一个COM容器(也就是包含COM对象的应用软件)就可以知道都有哪些接口可用了。

不幸的是，你还不能象询问一个VBX控件都有哪些属性那样询问一个COM对象它都支持哪些接口。相反，COM容器会列出一组作用于对象的预定义接口，直到得到正向反馈。“你能做这个吗?不能?好吧，那么那个怎么样?还是不能?那么其它的怎么样?”这意味着一个公司有可能创建只有该公司出品的COM容器才能使用的COM对象，并且如果它的设计者不打算公布它的接口……你可想而知那会怎样!

随之而来的另一问题是最初的接口经常没有包含所有需要的功能。比如，一旦你确定了一个你喜欢的COM对象，你必须创建它的一个实例以便使用它。这可能需要使用IClassFactory接口来完成。

然而，IClassFactory接口的设计者并没有考虑许可证的问题。OLE控件的创建者则要求你有一个许可文件支持控件在设计期间的使用。所以一个新的接口IClassFactory2出现了，它要有许可证的支持。这样，现在COM容器必须首先申请一个指向IClassFactory2接口的指针，并且只有当这个申请失败时，才申请IClassFactory接口。我对使用这种接口的容器代码的前景并不抱乐观态度，除非它找到使用真正能创建想要的COM对象的接口。

2.2 控件怪物

OLE1.0主要用于创建复合文档。OLE2.0试图在满足这个功能的同时，还增加了创建复合应用软件的功能。所以我们来谈谈OLE2.0控件，它通常叫做OCX。

对于那些从未接触过VBX的人来说，OCX看起来好象一种另人兴奋的想法。把一大把紧凑的、可重用的、可靠的功能加入到应用软件中真是一种伟大的思想。VBX就体现了这种思想，因为它公开了它的属性、事件以及方法。应用软件通过询问就可以知道这些信息。

OCX也公开了它的属性、事件和方法。这是通过IDispatch接口来完成的。这样OLE2.0的对象才通过大量代码实现了VBX已有的功能，并且还要有占1兆左右硬盘空间的四个DLL的支持。而VBX并不需要它们。每个使用这些DLL的应用软件必须发行它们，以免用户的机器上没有安装这些DLL。所以每一个应用软件包都要准备一个额外的软盘来发行它们。

不象VBX那样，应用软件通过询问就可以知道它的功能，COM对象的公共定义被移到了系统登记簿中(以前的Windows版本叫做注册数据库)。这有点违背面向对象的思想。如果你有过被迫重新安装Windows和你所有的应用软件的经历，你就会同我一样怀疑这种移动是否是聪明之举。然而，OCX(以及其它OLE2.0对象)在装入时，相对来说是很慢的，所以有必要将那些信息移到一个易获取的地方。早期OLE2.0对象的一个共同的问题是它们不能被移动：一个子目录的重命名或增加一个新硬盘会使你的复合文档不能再用，除非重新安装组件或手工修改登记簿。然而，用MFC编写的OLE2.0对象一般是自注册的——一种在我的书《Borland Pascal 7.0 Insider》中提及的技术——所以简单地在新的位置上运行或装入对象的实例就可以重注册它了。

VBX在装入时要比OCX快得多。但OCX却提供了对用户定义的方法的支持，而VBX不

支持它。使用 OCX 就不再需要 Action 属性来模拟用户定义的方法了。

在我写这本书时，MFC 还是能编写 OCX 的唯一类库。据说 Borland Delphi 的32位版会支持标准的16位 VBX，而 Microsoft 公司声明16位 VBX 不可能做为 OCX 标准的一部分。看来 Microsoft 是对的，因为 Delphi 2.0的发行版并没有包含这个特点。

如果你熟悉 MFC 的话，用 MFC 类编写 OCX 是件超前的事情。随着 Microsoft Visual C++ 1.52、2.2、4.0以及更高版本的发行，你会更容易地编写 OCX。你使用 Control Wizard 而不是 App Wizard 来创建一个工程，然后使用 Class Wizard 加入属性、事件和方法。当然这些东西的真正代码得由你来提供。

由于 VisualC++1.52只能生成16位代码，而 VisualC++2.2/4.0只能生成32位代码，所以如果你想要那些16位和32位的开发者都可以使用你的 OCX 你必须分别使用两种开发环境。幸运的是，你更换环境时不必改变代码。许多开发者使用 Visual C++2.2或4.0版开发控件，然后单独编译16位的版本。OLE2.0最初的承诺是调度层可以使32位应用软件能使用16位 OCX。但到现在我们没有看到这一点。

2.3 用你自己的眼睛观察吧!

Microsoft 公司总是有办法来证明它推出的软件组件是不可避免的和正确的。作为程序员，显然我们不能忽视那些新组件。并且我们当然应该使用那些新技术。但我们要以自己的眼光观察世界，不能人云亦云。每个冲浪者都知道，根本就没有什么“将来的浪潮”，不论波浪有多么大或惊心动魄，它总是一浪接一浪的。

第三章 VBX框架的回顾

在我的书《Windows Programming Power with Custom Controls》中，我开发了一种VBX框架。这个框架可以做为快速开发VBX的基础。象其它框架一样，它也存在优点和缺点。好的一面是它告诉你如何创建一个VBX或标准Windows控件的DLL(VBX就是DLL)。通常有些书中的例子程序是简化了的，只是为了演示某种技术和更易维护。与此相比，这里的VBX框架呈现了一个比较好的结构化代码的例子。而不好的一面是它只能说明如何创建3.0版本的VBX。这对于Visual Basic来说是可以的，但VisualC++和BorlandC++当前却只能使用1.0版的VBX。(当我写这本书时，VisualC++和BorlandC++正处于β测试阶段并且这两个平台的文档都没有说明这个限制。)而且，尽管Microsoft公司已经对OLE控件进行了大量宣传，但仍然没有迹象说明将VBX转换成OCX是容易还是困难。

在这一章，我将重新呈现给读者一种VBX框架，并且在原来基础上做了些修改。我们先来回顾一下VBX的结构。但这里有必要做了些简化，对VBX内部机制不了解的新手可以参考我以前写的那本书(或者去啃啃Microsoft公司控件开发工具箱中有关的技术资料)。

3.1 VBX的内部实现

VBX的核心是包含某种挂钩(hook)的DLL。这些挂钩使集成开发环境(IDE)能够识别那些VBX是做什么用的。挂钩所采用的是已知名字和调用顺序的函数形式。其中两个函数是VBINITCC()和VBTERMCC()。一个有VBX意识的集成开发环境，比如Visual Basic，当装入这个DLL时，将试图调用VBINITCC ()函数。接下来这个函数必须为每一个VBX支持的用户控件调用CDK(Control Deveiopment Kit，控件开发工具箱)函数VBRegisterModel()

挂钩函数VBINITCC()和VTTERMCC()使一个DLL成为VBX而与文件的后缀无关。但为了方便用户，几乎所有的VBX都有后缀VBX。

图3.1呈现了用来实现控件的VBX通常所具有的结构。

正如图中所示，最忙的部分是VISUAL. C。在我最初的框架中，我提供了另外一个部分MAIN. C。这个模块含有LibMain()(DLL的入口函数)和底层标准Windows窗口的窗口过程。由于大多数VBX不再基于底层窗口，所以我把这个部分去掉了。

VBXHELP. C没有做任何改动。我们将查看一下这部分的代码。它有点类似于菜谱。你不必对它做任何改动。MAIN. RC以前用于包含标准控件的对话框，现在不必这样做了。现在它包含了一些位图，这些位图用来做为在IDE的工具箱和About对话框模板中控件的外观显示。

让我们来进一步看看每个文件的内容吧。先从最重要的模块开始。

3.1.1 INTERNAL. H的内容

就象每一个成功的男人背后都有一个女人一样，INTERNAL. H是一个头文件，它并没有出现在VBX的结构图中，但它必须被每个C模块用#include语句包含进去。它包含了一组必须被包含的其它头文件和在辅助文件中定义的函数声明以及被模块使用的全局变量：

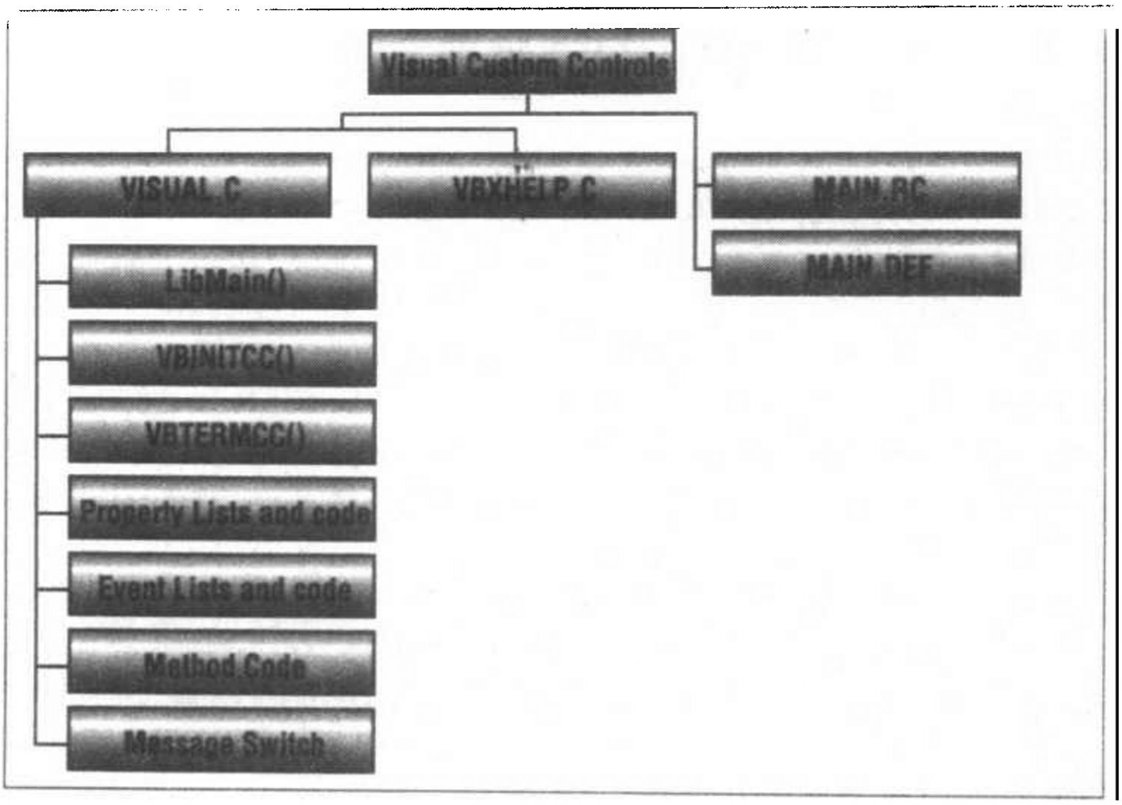

图3.1 包含一个控件的 VBX 结构

```
#include<windows.h>
#include<vbapi.h>
#include<stdlib.h>
extern HINSTANCE LibInstance;
HSZ far pascal GetAboutPropertyString(HCTL Control);
HWND far pascal PopupAbout(void);
void far pascal Register VbPopups(void);
void far pascal UnregisterVbPopup(void);
LPSTR far pascal HelpFile Name(void);
BOOL far pascal OnHelp(HWND Window, BYTE Help Type BYTE i,
                PROPINFO Properties [], PEVENTINFO Events []);
```

3.1.2 VISUAL.C 的内容

因为C最初是为一篇编译器设计的，所以一般来讲，应将结构和函数放在引用它们的结构和函数的前面，使C模块代码成为自顶向下的结构。所以我将自底向上地阐述 VISUAL.C 文件的内容。表3.1呈现了它所包含的一组函数。

VISUAL.C 文件中最后一个函数是 WEP()。WEP 代表 Windows 的退出过程。这个函数是所有 DLL 都需要的，并且当 DLL 被 Windows 下载时会被调用。但如果 DLL 没有提供 WEP()函数，则系统提供一个默认的 WEP 函数(经常是这样的)。当 DLL 被下载时，下面的函数必须被调用：

```
int far pascalexport WEP(int ExitType)
```

表3.1　　在 VISUAL. C 中定义的函数

函　　数	描　　述
LibMain()	DLL 的入口点。所有的 DLL 都要有这个函数，当 DLL 被装入时这个函数会被调用一次。
VBINITCC()	VBX 的入口点。这个函数会被 VBX 的使用者(如 Visual Basic)调用来注册 VBX 的控件。
CtlProc()	使用者的处理过程。它类似窗口过程，VBX 中的每个控件都必须具有一个这样的过程。
VBTERMCC ()	VBX 出口点。VBX 的使用者(如 Visual Basic)调用它使 VBX 在被下载前清除和释放所用的资源。
WEP()	DLL 的出口点。所有的 DLL 都要有这个函数。当 DLL 被下载(unload)时会调用这个函数。

```
{
UnregisterVbPopups();
}
```

下一个必需的函数是 LibMain()。它带有一个库实例句柄的参数，并且我们需要把这个句柄保存起来。以前版本的框架中，这个函数位于 MAIN. C 中。但 MAIN. C 除了包含这个函数外什么都没有。除了保存库实例句柄外，这个函数还要通过调用 RegisterVbpopups()函数(这个函数实际上是在 VBXHELP.C 中实现的)来注册 VBX 所需要的弹出式窗口：

```
HINSTANCE LibInstance=0;
int far pascal LibMain(HINSTANCE hInstance,WORD DataSeg,WORD HeapSize
LPSTR CommandLine)
{
if(HeapSize>0)Unlock Data(0);
LibInstance=hInstance;
Register VbPopups();
return TRUE;
}
```

当 VBX 被下载前，函数 VBTERMCC()将会被 IDE 或用户控件的使用者所调用。这时它所要做的工作是恢复在初始化时所做的任何事情。既使什么事情都不需要它去做，我们也必须要定义它：

```
VOID FAR PASCALexport VBTERMCC(void)
{
}
```

VBINITCC()看起来是只有一条执行语句的简单函数：

```
BOOL far pascal-export VBINITCC(USHORT Version,BOOL Runtime)
{
retum VBRegisterModel(LibInstance,&Model);
}
```

对 VBRegisterModel 的调用容易被人误解。我们传递给这个函数一个 MODEL 结构的地址，实际上这个结构本身才是最重要的(见图3.2)。

如图中所示，MODEL 结构包含指向两个列表的指针：属性列表和事件列表。而每个列表实际上又是一个指针的列表，并且指向标准属性和事件的指针是预定义的。此外，MODEL 结

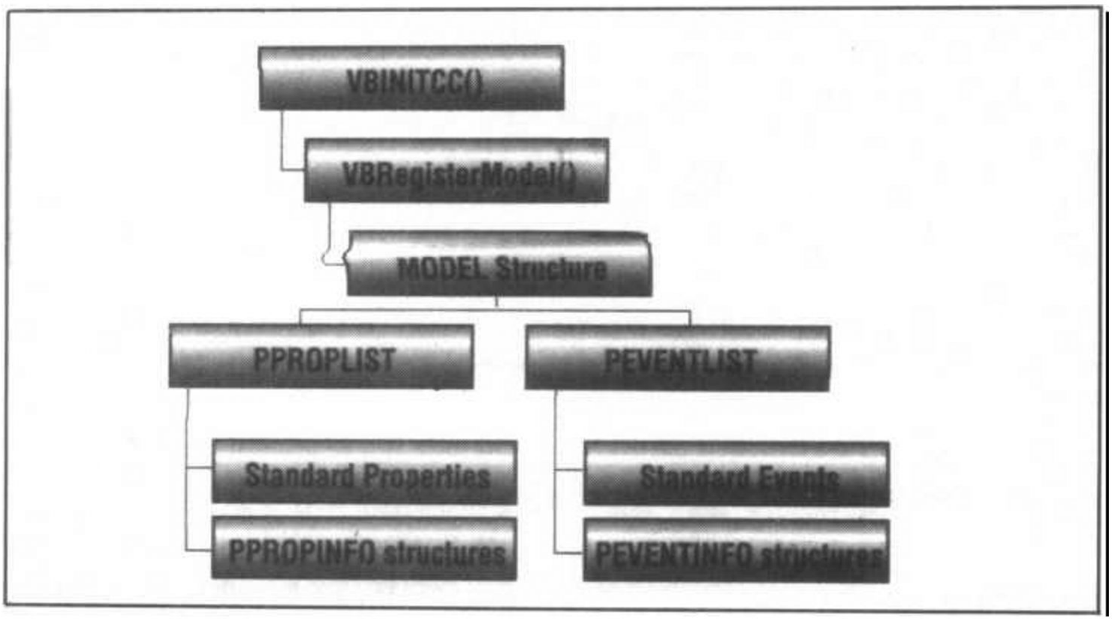

图3.2 MODEL结构的组织

构中还包含一个指向控件过程(类似于窗口过程)的指针和一些标记，这些标记进一步说明这个控件是如何显示以及怎样操作。

我简单地将Model变量存贮在控件的数据段上，所以不必浪费机时和代码空间去将它初始化。它是这样定义的：

```
MODEL Model=
    {
    VB300_VERSION,
    MODEL_fArrows|MODEL_fFocusOk,
    (PCTLPROC)CtlProc,
    CS_DBLCLKS,
    WS_BORDER,
    sizeof(VBDATA),
    8000,
    "Skeleton",
    "Skeleton",
    NULL,
    Properties,
    Events,
    0,
    0,
    (BYTE)-1
    93
    };
```

当然，其中的每个域都是非常重要的。但我还是将对这些域的讨论放到了第六章，在我讲到VBX Genie需要对这些域进行赋值时再讨论它们。现在，注意那两个字符串"Skeleton"，当这个框架被复制时，它们应该被新控件的名称所代替。第二、第四和第五个域分别是MODEL

风格、窗口类风格以及窗口的风格。如果你直接使用这个框架，你可能想改变这些风格。(VBX Genie 将会根据你的说明向这些域以及其它域中填入适当的值。)

第三个域提供了控件过程的地址：CtlProc()。当然，你可以对这个过程任意命名。事实上，如果你在一个 VBX 中包含多个控件的话，那就不只是重新命名的问题了。过程本身非常类似于常规的窗口过程，它接收与常规窗口过程同样的消息(以及一些新的消息)并且带有 Window，Message，wParam，lParam 参数。但是，它还有另一个参数：Control(控件)。这是一个控件本身的句柄。与标准窗口一样，系统中可以存在许多控件的实例，控件句柄是一个控件实例区别于其它实例的手段。

(1) 加入控件过程

在有些书的例子代码中作者通常是用内嵌式代码来编写窗口和控件过程的。我认为这种做法很令人讨厌。控件过程什么都不是，就是一个消息开关。每个到来的消息都应该被确认，将它的参数转换成适当的数据类型，激活正确的处理函数去做实际的工作。由于存在很多的 Windows 和 VB 消息，控件过程编写起来会很长。但代码是简单的，因为 switch 语句的每个 case 开关都是相似的：

```
long far pascal_export CtlProc(HCTL Control,HWND Window,USHORT Msg
                               USHORT wParam,long lParam)
{
long Error=0;
switch(Msg)
  {
  case WM_NCCREATE:
    OnNcCreate(Control,(LPCREATESTRUCT)lParam);
    break;
  case VBM_METHOD:
     return OnMethod(Control,Window,wParam,(void far*)lParam);
  case VBM_COMMAND:
     switch(HIWORD(lParam))
        {
       default: break;
        }
     break;
  case VBM_SETPROPERTY:
     if(OnSetProperty(Control,Window,wParam,lParam,&Error))
     return Error,
  case VBM_CHECKPROPERTY:
     if(OnCheckProperty(Control,Window,wParam,lParam,&Error))
     return Error;
  case VBM_GETPROPERTY:
     if(OnGetproperty(Control,Window,wParam,(LPVOID)lParam, &Error))
```

```
            return Error;
        case VBM_LOADPROPERTY:
            if(OnLoadProperty(Control,Window,wParam,(HFORMFILE)lParam,&Error))
            retum Error;
        case VBM_SAVEPROPERTY:
            if (OnSaveProperty (Control,Window,wParam,(HFORMFILE )lParam,&Error ))
            return Error;
        case VBM_GETPROPERTYHSZ;
            if(OnGetStringPreperty(Control,Window,wParam,(HSZ far* )lParam))
            return 0;
            else break;
        case VBM_INITPROPPOPUP:
            switch(wParam)
              {
            case IPROPINFO_Aboutreturn PopupAbout();
            }
            break;
        case VBM_HELP:
            if(OnHelp(Window,LOBYTE(wParam),HIBYTE(wParam),Properties,Events))
            return 0;
        case WM_DESTROY:
            WinHelp(Window,HelpFile Name(), HELP_QUIT, O);
            break;
        case WM_NCDESTROY:
            OnNcDesiroy(Control);
            break;
            }
    return VBDefControlProc(Control,Window,Msg,wParam,lParam);
    }
```

同窗口过程一样，有一个消息默认处理函数：VBDefControlProc()。你们中熟悉 MFC 的人会认识那些处理函数的名字，我这里借用了 MFC 消息处理函数的名字。

(2) 回过头来从顶部开始

从这开始我不再自底向上地讲，而是跳到文件的开头。我们会在文件的中间部分看到我们以前所讲的内容。这个文件以一个 #include INTERNAL. H 语句开始。这个头文件是这个工程所有模块都需要的。(可以使用编译器的预编译头特性，以节省编译时间。)接下来是一些 #define 语句和一个后面的代码将用到的变量：

```
    #include "internal. h"
    #define_segment(p)\
```

```
((HANDLE)(((unsigned long)(void far*)(p))>>16L))
#define_offsetin(struc,fld)(USHORT)&(((struc*)0)->fld)
#define VBERR_BADINDEX 381
static long Boolean [2] = {0, -1};
```

segment()的宏定义是我从 Microsoft 公司的 CDK 例子中拷贝过来的，但却从来没有实际使用过。当要出版第一个框时，我有机会使用 segment()宏定义3，但使我感到困惑的是我发现它并不象最初想象的那么有用。

Boolean 数组使在 C 对 TRUE（1）和 FALSE(0)定义与 Visual Basic 对它们的定义之间进行转换变得非常容易。

接下来是框架的结构类型定义，它用于存贮特定的控件数据，换句话说，就是属性的内部表示。在这个框架中，这个结构类型并没有用来定义某个成员变量。

```
typedef struct
 {
UNIT: 16;
} VBDATA, far*LPVBDATA;
```

接下来是用户属性结构。在这个框架中，只有一个“(关于)(About)”的属性：

```
PROPINFO Property_About=
 {
 "(About)",
DT_HSZ|PF_fGetMsg|PF_fNoRuntime W|PF_fGetHszMsg,
0, 0, 0, NULL, 0
};
```

如果需要更多的用户属性，可以相应地多定义一些 PROPINFO 结构(一般情况下是这样的)。指向所有用户属性结构的指针以及预定义标准属性的值被用来初始化那些指针的数组，这个数组的指针包含在 Model 结构中：

```
PPROPINFO Properties [] =
   {
  PPROPINFO_STD_NAME,
  PPROPINFO_STD_INDEX,
  PPROPINFO_STD_ENABLED,
  PPROPINFO_STD_HELPCONTEXTID,
  PPROPINFO_STD_LEFT,
  PPROPINFO_STD_TOP,
  PPROPINFO_STD_WIDTH,
  PPROPINFO_STD_HEIGHT,
  PPROPINFO_STD_PARENT,
  PPROPINFO_STD_LAST,
  &Property_About,
  //这儿可以加入你额外的 PROPINFO 地址
  NULL
```

```
};
```

NULL 作为列表的最后一个表项结束了这个列表。

用有意义的标识符来给列表的索引命名是很有帮助的。以前要么用#define 语句来定义它们，但这会带来一个问题：调试器并不能识别它们；要么增加属性，这可能意味着要手工编辑所有的#define 语句。但幸运的是，已经有人使用匿名枚举技术来解决这个问题：

```
typedef enum
    {
    IPROPINFO_STD_NAME,
    IPROPINFO_STD_INDEX,
    IPROPINFO_STD_ENABLED,
    IPROPINFO_STD_HELPCONTEXTID,
    IPROPINFO_STD_LEFT,
    IPROPINFO_STD_TOP,
    IPROPINFO_STD_WIDTH,
    IPROPINFO_STD_HEIGHT,
    IPROPINFO_STD_PARENT,
    IPROPINFO_STD_LAST,
IPROPINFO About,
//这儿可以增加你另外一些属性索引
IPROPINFO_End
};
```

这里并不需要 IPROPINFO_End，但它很容易使你注意到你定义了多少属性。并且它也可以使你很容易在列表尾部加入属性而不必担心是否会忘记要加上逗号。

(3) 支持的事件、方法和属性

讲到这儿，如果你有用户事件，你就要初始化 EVENTINFO 结构。接下来就是由指向这些结构的指针、标准事件的标识符以及事件的枚举索引组成的数组：

```
//在这儿定义用户事件
PEVENTINFO Events [] =
    {
  PEVENTINFO_STD_CLICK,
  PEVENTINFO_STD_DBLCLICK,
  PEVENTINFO STD LAST,
  //在这儿加入你自己的 EVENTINFO 地址
  NULL
  };
typedef enum
    {
  IEVENTINFO_STD_CLICK,
  IEVENTINFO_STD_DBLCLICK,
```

```
IEVENTINFO STD LAST,
//在这儿加入你自己的事件索引
NULL
} EVENTSIX;
```

注意，你可能要改变标准事件的列表。

OnNcCreate()和 OnNcDestroy()分别用来分配和释放那些控件在它的生存期一直要使用的资源：

```
static void near pascal OnNcCreate(HCTL Control,LPCREATESTRUCT Create)
  {
  LPVBDATA VbData=(LPVBDATA)VbDerefControl(Control);
  //在这儿分配内存或系统资源
  }
static void near pascal OnNcDestroy(HCTL Control)
  {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  //在这儿释放内存或系统资源
  }
```

VBX 用户所激活的任何标准方法(在 VBX 中并没有用户方法)都将生成一个 VBM_METHOD 消息。这个框架将这些消息传递给默认的处理：

```
static long near pascal OnMethod(HCTL Control,HWND Window,USHORT Method,void far* Args)
    {
    LRESULT Result=0;
    switch(Method)
      {
      default:
      Result = VBDefControlProc (Control, Window, VBM_ METHOD, Method, (long) Args);
    }
    return Resuit;
    }
```

根据你在 PRPPINFO 结构中所设置的标志，属性既可以是自动被设置的又可以是通过消息来设置的。对属性值的获取也是这样的。在这个框架中我们只提供了一个代码架子：

```
static BOOL near pascal OnSet Property(HCTL Control,HWND Window,USHORT PropertyIX,long Value,long far* Error)
  {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  switch(PropertyIX)
  {
  default:
```

```
        return FALSE;
        }
    }
static BOOL near pascal OnGetProperty (HCTL Control,HWND Window,USHORT Prop-
ertyIX,
        long Value, long far*Error)
      {
      LPVBDATA VbData= (LPVBDATA) VBDerefControl (Control);
      switch (PropertyIX)
      {
      default:
        return FALSE;
        }
    }
```

除了被设置和获取外，属性在被设置前还可以被检验：

```
      static BOOL near pascal OnCheckProperty(HCTL Control,HWND Window,USHORT
PropertyIX
      long Value,long far*Error)
      {
      LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
      switch(PropertyIX)
        {
        default:
          return FALSE;
        }
    }
```

一些属性（如颜色）可能需要用字符串来表示以便使它们可以出现在 IDE 的属性窗口中。属性“(关于)”（它只是一个假定的属性）就是一个这样的属性。（单击“(关于)”属性会弹出控件的关于消息框。）对这种字符串的操作需要由下面的函数来完成：

```
static BOOL near pascal OnGetStringProperty(HCTL Control, HWND Window,
        USHORT PropertyIX,HSZ far*Value)
      {
      LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
      switch(PropertyIX)
        {
        case IPROPINFO_About:
          *Value=GetAboutPropertyString(Control);
          break;
        default:
          return FALSE;
```

```
    }
  return TRUE;
  }
```

最后，属性可以从磁盘上被装入，也可以存贮在磁盘上。这些工作由下面的函数来完成：

```
static BOOL near pascal OnLoadProperty (HCTL Control, HWND Window, USHORT PropertyIX,
HFORMFILE FormFile, long far*Error)
    {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  switch(PropertyIX)
      {
    default:
      return FALSE;
  }
}
static BOOL near pascal OnSaveProperty (HCTL Control, HWND Window, USHORT PropertyIX,
HFORMFILE FormFile,long far*Error)
    {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  switch(PropertyIX)
      {
      default:
      return FALSE;
    }
  }
```

3.1.3 VBXHELP. C 的内容

一个框架包含这个文件是很自然的事，并且我们从来都无需修改它。这个文件以 #include 语句开始：

```
#include "internal. h"
```

接下来是一个小函数，它是在响应 VBM_GETPROPERTYHSZ 消息时被激活的，并且返回属性"关于"的字符串。CDK 函数 VBCreateHSZ()接受 C 风格的字符串并且返回一个 VB 字符串类型的句柄：

```
HSZ far pascal GetAboutProperty String(HCTL Control)
  {
  return VBCreateHsz(segment(Control), "Click here for About→");
  }
```

关于对话框的对话框过程是很普通的一种：

```
BOOL_export FAR PASCAL AboutDlgProo(HWND Dialog, WORD msg,
```

```
        WORD wParam,long lParam)
    {
  switch(msg)
   {
    case WM_INITDIALOG:
         break;
    case WM_ COMMAND:
      switch(wParam)
        {
        case IDOK:
          EndDialog(Dialog,IDOK),
          break;
        default:
          return FALSE;
          }
        break;
      default:
        return FALSE;
        }
    return TRUE;
    }
```

但是，由于各种原因，你不能在VBX中直接调用DialogBox()这个Windows函数。而是创建一个不可见的弹出式窗口，在这个窗口中运行这个对话框。这就意味着你需要一个这个弹出式窗口的窗口过程：

```
  LONG_ export FAR PASCAL AboutPopupWndProc(HWND Window,USHORT Mes-
sage,
        USHORT wParam,LONG lParam)
    {
  switch(Message)
    {
  case WM_SHOWWINDOW:
        if(wParam)
          {
        Show Window(Window,SW_HIDE);
        PostMessage(Window,WM_USER,0,0);
        return 0;
        }
  case WM_USER:
        VBDialogBoxParam(LibInstance,"About",AboutDlgProc,NULL);
        return 0;
```

```
    }
  return DefWindowProc (Window, Message, wParam, lParam);
  }
```

至少应感谢编译器设计方面的进步，我们不必获取对话框过程的“过程实例”了，我们可以直接传递它的地址。

为了使“关于”对话框真正显示，接下来要调用 PopupAbout()函数，它创建那个弹出式窗口并且运行“关于”对话框：

```
static char AboutPopupClass [] = "AboutPopup";
HWND far pascal PopupAbout(void)
     {
     return CreateWindow
       (
       AboutPopupClass,
       NULL,
       WS_POPUP,
       0,0,0,0,
       NULL,
       NULL,
       LibInstance,
       NULL
       );
     }
```

你还记得在 VBINITCC()中被调用的 Register VbPopups()函数吗?好，这就是它的定义：

```
static WNDCLASS Class=
    {
    0,
    (WNDPROC)AboutPopupWndproc,
    0,
    0,
    0,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
    };
void far pascal Register VbPopups(void)
    {
    Class. hInstance=LibInstance;
    Class. lpszClassName=AboutPopupClass;
```

```
    RegisterClass(&Class);
    }
```

由于 WNDCLASS 结构被预先初始化了，我们只需填入实例和类名，并注册这个类就可以了。

正如我们所看到的，当 WEP()被调用时，那个弹出式窗口需要被取消：

```
viod far pascal Unregister VbPopups(void)
 {
UnregisterClass(AboutPopupClass,LibInstance);
}
```

一般地，VBX 的帮助文件的名字是从 VBX 名字派生出来的，只不过文件的后缀是 HLP 而不是 VBX。所以 HelpFileName()函数提供了一个用来派生的编程方法：

```
LPSTR far pascal HelpFileName(void)
  {
  static char Pathname [_MAX_PATH] = {0};
  if(!Pathname[0])
      {
      char Drive [_MAX_DRIVE];
      char Dir [_MAX_DIR];
      char FName [_MAX_FNAME];
      char Fxt [_MAX_EXT];
      GetModuleFileName(LibInstance,Pathname,_MAX_PATH);
      _splitpath(Pathname,Drive,Dir,FName,Ext);
      _makepath(Pathname,Drive,Dir,FName"HLP");
      }
      return Pathname;
      }
```

当 VBX 用户以选择 IDE 的菜单或按 F1键的方式请求帮助时，VBM_HELP 消息会被发送到控件过程：

```
BOOL far pascal OnHelp(HWND Window,BYTE Help Type,BYTE i,
PPROPINFO Properties[],PEVENTINFO Events[])
  {
  BOOL Resuit=FALSE;
  switch(Help Type)
     {
    case VBHELP_PROP:
     if(Properties[i]<PROPINFO_STD_LAST)
        {
       WinHelp(Window,HelpFileName(), HELP_KEY,
         (DWORD)(LPSTR)Properties [i] ->npszName);
       Resuit=TRUE;
```

```
        }
      break;
    case VBHELP_EVT:
      if(Events[i]<PEVENTINFO_STD_LAST)
        {
        WinHelp(Window,HelpFileName(), HELP_KEY,
          (DWORD)(LPSTR)Events [i] ->npszName);
        Kesuit=TRUE;
        }
      break;
    case VBHELP_CTL:
      WinHelp(Window,HelpFileName(), HELP_CONTENTS, 0);
      Result=TRUE;
      break;
    }
  return Resuit;
  }
```

只要你的帮助文件提供了与你的控件的属性名和事件名相匹配的主题，这个函数就能自动提供了上下文敏感的帮助。

3.1.4 MAIN.RC 文件的内容

这个资源描述文件提供了四种资源：工具箱按钮的位图、“关于”对话框的模板、用在“关于”对话框中的图标以及一个 VERSIONINFO(版本信息)资源。位图资源本身必须以另外一个 BMP 文件存贮。尽管你不必亲自动手去编写这个文件(你完全可以使用 Visual C++中的 App Studio 或 Boriand C++中的 Resourse Workshop)，我还是要呈现一下这个文件的内容：

```
#include<windows.h>
#include<ver.h>
8000 BITMAP MOVEABLE PURE "8000.BMP"
8001 BITMAP MOVEABLE PURE "8001.BMP"
8003 BITMAP MOVEABLE PURE "8003.BMP"
8006 BITMAP MOVEABLE PURE "8006.BMP"
ABOUT DIALOG 41, 45, 192, 125
STYLE DS_MODALFRAME/WS_POPUP/WS_CAPTION
CAPTION "About Skeleton Custom Control"
FONT 8 "MS Sans Serif"
BEGIN
    CONTROL "Skeleton Custom Control", 0, "STATIC", SS_CENTER|WS_CHILD
              |
              WS_VISIBLE, 0, 0, 190, 8
```

```
        CONTROL"by Paul S. Cilwa",0,"STATIC",SS_CENTER|WS_CHILD|WS_VISI-
                BLE,
                0, 16, 190, 8
        CONTROL "MAIN" 0, "STATIC", SS_ICON|WS_CHILD|WS_VISIBLE, 87, 29.
18, 20
        CONTROL "Version 1. 0. 0. 0", 0, "STATIC", SS_CENTER|WS_CHILD|WS
                VISIBLE,
                0, 59, 190, 8
CONTROL "@1995 Paul S. Cilwa All Rights Reserved", 0, "STATIC", SS_CENTER|
        WS_CHILD|WS_VISIBLE, 0, 79, 190, 24
CONTROL "OK", IDOK, "BUTTON", BS_PUSHBUTTON|WS_CHILD|WS_VISIBLI
WS_TABSTOP, 74, 102, 45, 15
      END
      MAIN ICON "MAIN. ICO"
      I VERSIONINFO LOADONCALL MOVEABLE DISCARDABLE
      FILEVERSION, 1, 0, 0, 0
      PRODUCTVERSION, 1, 0, 0, 0
      FILEOS VOS_WINDOWS16
      FILETYPE VFT_DLL
      BEGIN
        BLOCK "StringFileInfo"
        BEGIN
          BLOCK "040904E4"
            BEGIN
                    VALUE "CompanyName", "The Coriolis Group\000"
                    VALUE "FileDescription", "Skelet on Custom Control for Visual Basic
\000"
                    VALUE "File Version", "1. 0. 0. 0\000"
                    VALUE "InternalName", "Skeleton\000"
                    VALUE "LegalCopyright", "@1995 Pa ul S. Cilwa All Rights Reserved
\000"
                    VALUE "OriginalFileName", "Skeleton. vbx\000"
                    VALUE "ProductName", "Skeleton Custom Control\000"
                    VALUE "ProductVersion", "1. 0. 0. 0\000"
                    VALUE "Comments", "000"
            END
        END
      END
```

3.1.5 MAIN. DEF 文件的内容

最后，我们来看看模块定义文件：

```
LIBRARY Skeleton
DESCRIPTION "1996 Paul. S Cilwa All Rights Reserved"
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE SINGLE
HEAPSIZE 2048
```

如果你使用的是 Microsoft 连接器，你可以加入下面一行：

```
EXPORTS
  WEP RESIDENTNAME
```

RESIDENTNAME 选项保证 WEP()函数的名字将会一直可用。但 Borland 连接器并不认识这个关键字。如果没有它也不会碍什么事的。

3.2 小结

以上简略地介绍了这个 VBX 框架。因为我想让你理解 VBX 的总体结构，所以我特意介绍了一些细节，特别是各种结构的内容以及它们所使用标志的含义。以后我们将要更加详细地讲解这些细节，但你不会再感到它们是那么可怕了，因为你已经在一定程度熟悉它们了。

下一章我们将以完全不同的角度来讨论 VBX 的结构。你在这一章所学到的知识会帮助你理解它。让我们开始吧！

第二部分　一個VBX類庫

第四章　创建一个与软件商无关的GDI类库

在前几章中，我们回顾了用C创建用户控件的一些基本知识，还介绍了一个用于创建VBX用户控件的框架。现在，我们要转向C++了。

4.1　C的缺点

在我们开始前，你肯定会问：为什么非要用C++来创建用户控件呢?用C来编写会有什么麻烦吗?噢，我们当然不会遇到什么麻烦。但确实存在着一些因素使C编写的VBX变得很脆弱。比如，CDK所涉及的结构中有许多域，而这些域很容易被误用。当然，使用一种框架来编写VBX可以解决一部分问题。但C++的强制类型检查会完全解决这些问题。

同样，许多CDK代码都是重复的。C处理这种问题的办法是复制框架，而C++有更好的办法：把那些重复的代码封装在类库里。这样你就可以继承它，而不是复制它了。

此外，一个真正的框架应该为所有可能的消息实现消息处理器(message handler)。但除了160个Windows消息外，VB还增加了41个自己的消息。这就意味着一个框架中会有相当多的消息处理器，而对于每个从该框架中派生出来的有特定用途的VBX来说，绝大部分消息处理器都要从框架中删除掉。如果用C++，所有的消息处理器都可以定义在类库中，对于那些不需要特别处理的消息，你可以继承相应的消息处理器；而对于其它的消息，你只需重载它们的处理器就可以了。

4.2　C++的优点

好，既然用C编写用户控件有缺点，那么用C++编写就有优点吗?

事实上就是如此。其实，将CDK中定义的结构略作修改就可以构成C++的类。你应该知道，任何C的结构都可以作为C++类的基类。CDK中包含许多组函数，几乎所有这些函数都以这样或那样的结构做为第一个参数。所以，如果以某个结构派生出一个类，那么以这个结构做为第一个参数的函数就可以作为这个类的成员函数。这种划分是很容易、很自然的事。

下面，我们再来看看那些重复的代码。当它们被隐藏在类库中时，你就不必担心它们了。你只需编写你所需要的特定的代码就可以了。实际上，这也是C++最大的优点。使用C++你可以将注意力集中在编写特定代码上，避免了编写重复代码的乏味工作。这只是第二大优点。第一大优点是OCX(即OLE控件)必须用C++来编写。通过用C++编写VBX，我们会发现我们将很容易操纵我们的代码，使它们既可以用于VBX又可以用于OCX。

4.3 我们的目标

无论怎么说，编写一个 VBX 都应该是一件不费力的事情。你不会希望上下翻屏，在大量生成的代码中寻找插入你自己代码的位置。当然，在生成的代码中通常会有一些要被删除掉。

在 Visual Basic 中有一个向应用程序加入用户 C++组件的选项。当你在编写一个 VB 应用程序时，你会想"嗯，这组我要编写的函数会对用户控件非常有用"，而且你会很容易完成这些函数的编写。

一个程序员应该在很短的时间内就可以编写出可以使用的 VBX。VBX 的 C++类库可使程序员做到这一点。而且当你通过生成新派生类的框架说明和方法来向 VBX Genie 中加入更强的辅助功能时，编写 VBX 就更加省事了。

可奇怪的是，为了创建这样一个工程，必须首先决定：我们是用 ObjectWindows 还是用微软基本类库(MFC)?新类库在理想的情况下应不依赖于任何一种环境。但也存在风格上的问题：比如，我们调用哪种方法?是 OnPaint()，还是 EvPaint()?我们是按照 C++风格(在多数情况下)使用引用型参数呢?还是模仿 MFC，按照旧的 C 风格，使用指针参数?无论我们选择哪种风格：ObjectWindows 或是 MFC，其它类库的用户都会发现我们的函数调用习惯是很奇怪的。

接下来还有一个命名的问题：类的名字是用"C"(MFC 中是这样的)还是以"T"(ObjectWindows 中是这样的)开头?最初，Borland 公司的 ObjectWindows 是为 Windows 版的 Turbo Pascal 编写的。在 Turbo Pascal 中，对象是使用 **Type** 语句来说明的；所以，对象被看作是"类型(Type)"并且以前缀字母"T"来标识。当 ObjectWindows 被转换到 C++时，这个习惯被保留了下来。

由于 MFC 一开始就是 C++类库，而且 C++的类是使用 **class** 语句来说明的，前缀字母当然就是"C"了。但是，随着 OLE 和它的接口类的出现，一些类使用前缀"T"。

最初 C++的习惯是没有前缀字母，比如 iostream 类等等。现在，如果你要在命名时加入前缀，数据类型要以小写字母开始。后面使用的 C++习惯的好处就是类的名字会与内部 C 数据类型(如 **int**,**float** 和 **char**)有相似之处。由于我经常将类实现得比较完全，并且尽量使用重载操作符，包括释放操作符，所以我比较喜欢这种习惯。

还有另一个需要面对的问题。我们需要使用 MFC 和 ObjectWindows 都支持的两个类：一个字符串类和一个 GDI 子类。幸运的是：我已经编写出了这两个与软件商无关的类，所以我只需在工程中使用它们就可以了。但你肯定想知道它们是如何设计的；毕竟，它们是你所要设计的类库的一部分。所以，在我们开始建造 VBX++类库前，我们要在美妙的 Windows GDI 世界中做一次短暂的旅行。

4.4 创建一个与软件商无关的 Windows 类库

我在许多地方教过 Windows 程序设计。有些学生学习用 MFC 编程，而有些学生则使用 ObjectWindows。虽然这两个类库在逻辑上是相似的，但它们之间有太多术语上的差异，所以两种代码是很难相互交换的。这是个很丢脸的事，因为几乎所有的学生都问我，*如何编写他们的* Windows 代码才能不依赖于任何一家软件商呢?

当然，当我打算为VBX编写一个类库时，也遇到了这个问题。我应该编写依赖于MFC的代码吗？如果是这样，Borland程序员就不会使用它；但如果我使用ObjectWindows，Visual C++程序员就会被忽略。

如果存在一个与软件商无关的、综合了两者的长处、消除了两者的弱点的Windows类库不就好了吗？

在这一章我们将从介绍一个小型的GDI类库来开始这方面的努力。Windows GDI子系统在MFC中和OWL(ObjectWindowsLibrary)中的功能是最弱的。我们要创建的各个类都列在表4.1中了。这个表列出了每个类的名字和定义这个类的文件名。

表4.1　　用来实现小型GDI库的类

类　名	文件名	描　述
Point	POINT. CPP	提供表示坐标点的内部结构和函数
size	POINT. CPP	用于表示rectangle(矩形)对象的大小
rectangle	POINT. CPP	提供表示rectangle的内部结构和函数
colorbyte	COLOR. CPP	存贮用于表示一种彩色的三基色所需要的信息
color	COLOR. CPP	提供使程序员很容易地设置彩色三基色值的所有的结构和函数
brush	BRUSH. CPP	提供用于实现使用实彩色和自定义斜线模式的自定义刷子
pen	PEN. CPP	提供用于实现自定义画笔的函数
devicecontext	DEVICE. CPP	提供用于为这个GDI库建立设备无关性的支撑结构和函数
paintcontext	DEVICE. CPP	建立一个类，用于控制与设备无关的作图
clientcontext	DEVICE. CPP	建立一个类，用于控制在用户区域的作图

现有的GDI类中最大的问题是每种GDI类都与它们的相应类库紧密相联。这意味着，如果你因为某种原因不能在工程中使用MFC或OWL，你可能也不能用GDI函数。这个问题在为VBX编写一个C++类库以前必须解决。尽管Microsoft公司声称使用MFC可以编写DLL(VBX就被实现成DLL)，但这么做是非常复杂的，所以我从未教给任何人。谁会去这么做！

所以我决定来啃这块硬骨头，建造一个GDI类库。我并没有重复那些Microsoft公司和Borland公司已经做得比较好的表层封装。尽管它还不是完整的(谁有那么多的时间？)它所实现的目的只是对Windows SDK的改进，而不是创建C++类。毕竟，如果你还没有体验到C++的好处，他们这样做又有什么关系呢？

4.5　GDI的基本知识

Windows GDI由多个对象类型组成。它的核心是设备环境(device context)。设备环境在C代码中是通过句柄(handle)来获取的，通常缩写成hdc。设备环境代表应用程序可能会在上面作图的那些东西：如应用程序窗口的用户区域、整个屏幕、打印机的一页、甚至一个位图或一块内存。虽然不同的设备环境类型将有不同的获取方式，但在设备环境中作图的技术是不变的。这意味着在屏幕上作图的代码也可以用于图形打印，只需作图函数以相应设备环境作为参数就可以了。

设备环境中一直维护着三种作图工具：刷子、画笔和字体。当你刚获取一个设备环境时，

这个设备环境中已经有了这三个工具。几乎所有的有关 Windows 程序设计的资料都告诉我们，当你要把一个设备环境返回给系统时，如果你已经在设备环境中选择了新的工具，你必须首先恢复设备环境中原来的画笔、刷子和字体。其实并非如此。虽然你千万不要将带有自定义的画笔、刷子和字体的设备环境返回给系统，但可以将带有库存对象的设备环境返回给系统。

画笔、刷子和字体可以按照具体说明去创建，也可以从公用对象库中借出来，比如黑色画 笔或系统字体。每一种库存对象在 Windows SDK 中是通过特殊类型句柄来管理的，而在这里都应该用单独一个 C++类来表示。虽然并非所有的设备环境都有相联系的位图，但你可以象处理画笔和刷子那样向设备环境中加入一个位图。所以位图也需要用单独一个类来表示。

最后，虽然严格地说象颜色(color)、点(point)、矩形(rectangle)这些部分并不属于 GDI，但它们最好还是用类来表示。因为其它类需要引用这些部分，我们将从那里开始。

当我编写一个类时，我通常用一对文件来包装它。类的说明放在头文件中，当然通常使用一些 **inline** 语句。而类的定义部分放在 CPP 文件中。但理想的做法是用两个文件 PORTAGDL.H 和 PORTAGDL.LIB 来包装这个库文件。

在我们浏览代码时，我会提醒你什么时候我们要从头文件跳到代码文件。但有一点你要清楚：如果它不是一个 **class** 语句或 **inline** 函数，你必须查看相应的 CPP 文件。

我同时使用一个技巧以避免一个头文件被包含两次。我用下面的编译预处理指令将头文件括起来：

```
#ifndef_MYFILE_H
#define_MYFILE_H
/
/
#endif
```

这里“_MYFILE_H”是从文件名派生出来的。我也确保了每个头文件都包含有 **#include** 指令将那些它引用的其它头文件包含进来。这样使用我的类库的程序员从不用担心使用 **#include** 语句的顺序。

4.6 您的出发点

这个小型 GDI 库中第一个类处理的是坐标系统。它们是在 POINT. CPP 中实现的，包括点、大小、矩形。为了能容易与 SDK,MFC,OWL 共同使用，这些类是从标准 POINT 和 SIZE 类派生出来的。我们将从定义 point 类和实现它的构造函数和操作符函数开始。然后，我们会创建 size 类和 rectangle 类。

下面是 point 的全部定义：

```
class size;
class point: public tagPOINT
  {
  public:
    point(int X=0,int Y=0){x=X,y=Y};
    point(LPARAM lParam);
```

```
    point(const point &. Original);
    BOOL operator==(const point &. Test)const;
    BOOL operatorl=(const point &. Test)const;
    point &. operator=(const point &. Original);
    point &. operator+=(const size Size);
    point &. operator=(const size Size);
    friend point operator+(const point Point,const size Size);
    friend point operator-(const point Point,const size Size);
    size operator-(const point Point)const;
  };
```

除非我们确实从标记名(tagname)而不是从结构名(**struct** name)派生出新类，否则使用 Microsoff 公司的编译器时就会遇到麻烦。因此，我们在类的定义中使用了名字 tagPOINT。还要注意，由于 point 类中的一些方法以 size 对象作为参数或返回值，所以对 size 类做了预说明。我们在后面定义 size 类。

再强调一次，这个类是最基本的、不完全的，如果你需要更多的算术操作符(总有一天你会需要的)，你再增加它们。我这里只提供了 ObjectWindows 的 TPoint 类中相应的操作符。

当决定在类中应包含哪些操作符和那些操作符的意义时，你一定要考虑到最终使用这些类的程序员的需要。在这里，两个点相加生成一个矩形而不是一个点，所以这个操作符是在 rectangle 类中实现的。另一方面，将一个点加上一个大小，逻辑上将产生一个新点；一个点减去另一个点将产生一个大小。

4.6.1 构造函数和操作符函数的实现

这三个构造函数(有两个在 POINT.CPP 实现)不能在成员初始部分初始化 x，y，因为这两个属性是从基类中继承过来的。而必须用标准赋值语句来完成：

```
point::point(LPARAM 1Param)
{
x=LOWORD(1Param);
y=HIWORD(1Param);
}
point::point(const point &. Original)
{
x=Original.x;
y=Original.y;
}
```

等价测试是直接的，不等价测试只是调用等价测试：

```
BOOL point::operator==(const point &. Test)const
{
return((x==Test.x)&&(y==Test.y));
}
BOOL point::operator !=(const point &. Test)const
```

```
{
return ! (*this==Test);
}
```

三个赋值操作符(一个是普通的赋值,另外两个是算术操作符)按照要求的模式,返回* this,即对象本身的引用:

```
point &. pont::operator=(const point &. Original)
{
x=Original. x;
y=Original. y;
return *this;
}
point &. point::operator+=(const size Size
{
x+=Size. cx;
y+=Size. cy;
return *this;
}
point &. point ::operator-=(const size Size)
{
x-=Size. cx;
y-=Size. cy;
return *this;
}
```

两个全局(友元)操作符创建了一个暂时对象以供返回;并且对这个中间对象使用算术赋值操作,所以加减逻辑不必重复了:

```
point operator+(const point Point,const size Size)
{
return point(Point)+=Size;
}
point operator-(const point Point,const size Size)
{
return point(Point)-=Size;
}
```

剩下的操作符可以使一个点减去另一个点,返回一个 size 对象:

```
size point ::operaor-(const point Point)const
{
return size(x-Point. x,y-Point. y);
}
```

4.6.2 定义 size 类

size 类的复杂程度方面与 point 类差不多。它在定义 point 类说明前需要预说明一下。下面

是 size 类的全部说明：

```
class size: pulic tagSIZE
{
pulic:
size(int CX=0,intCY=0){cx=CX;cy=CY;}
size(LPARAM1Param);
size(const size & Original);
BOOL operator==(const size & Test)const;
BOOL operator!=(const size & Test)const;
size & operator=(const size Size);
size & operator+=(const size Size);
size & operator-=(const size Size);
size operator+(const size Size)const;
size operator-(const size Size)const;
}
```

由于 size 类的代码与 point 类非常相似，所以我不想用它实现代码来烦扰你了。

4.6.3 介绍 rectangle 类

rectangle 类从 tagRECT 中继承了四个属性(而不是两个属性)：

```
class rectangle: public tagRECT
  {
  public:
   rectangle(int aLeft=0,int aTop=0,int aRight=0,int aBottom=0);
   rectangle(point TopLeft,point BottomRight);
   rectangle(point Point,size Size);
   rectangle(RECT Rect);
   int GetWidth(void)const;
   int GetHeight(void)const;
   long GetArea(void)const;
   point GetTopLeft(void)const;
   point GetBottomRight(void)const;
   BOOL lsEmpty(void)const {return IsRectEmpty(this);}
   BOOL operator==(const rectangle & Rect)const;
   BOOL operator!=(const rectangle & Rect)const;
   BOOL IsIn(const rectangle & Rect)const;
   BOOL Contains(const rectangle & Rect)const;
   BOOL Contains(const point & point)const;
   rectangle & operator-(const rectangle & Original);
   rectangle & operator+=(const rectangle & Rect);
   rectangle & operator+=(const size & Size);
```

```
rectangle & operator-=(const size & Size);
};
```

rectangle 类有 left，right，top，bottom 四个属性，因此它的构造函数中简单地使用了赋值语句，例如：

```
rectangle::rectangle(int aLeft. int aTop,int aRight,int aBottom)
  {
  left=aLeft;
  top=aTop;
  right=aRight;
  bottom=aBottom;
  }
```

但你可能对有复制操作的构造函数感到奇怪：

```
rectangle::rectangle(RECT Rect)
  {
  CopyRect(this,&Rect);
  }
```

你可能会问“CopyRect 是从哪儿来的?如果它是成员函数，我没有见到说明过它呀!”。噢，从某种方式上说，它是一个成员函数，一个继承的函数。可 tagRECT 不是一个结构吗?是这样的。但用 C 编写类似面向对象程序的方法是编写操作相应数据结构的函数。如果你在 Windows API 联机求助中查找“rectangle 函数”，你会看到 Microsoft 公司已经为操作 rectangle(更确切地说是 tagRECT)提供了几个函数。我们也可以使用它们。

所以等价和不等价测试可以利用另一个 Windows API 函数 EqualRect()；

```
BOOL rectangle::operator==(const rectangle & Rect)const
  {
  return EqualRect(this,&Rect);
  }
BOOL rectangle::operator!=(const rectangle & Rect)const
  {
  return !EqualRect(this,&Rect);
  }
```

成员函数 GetTopLeft()和 GetBottomRight()返回 point 类对象。它们的实现跟你想象的一样。编写这两个函数的主要原因是为了实现其中一个 Contains 函数。如果一个 rectangle 对象完全位于这个矩形内部，这个函数返回 TRUE：

```
BOOL rectangle::Contains(const rectangle & Rect)const
{
return Contains(Rect.GetTopLeft())&&Contains(Rect.GetBottomRight());
}
```

这个函数调用了另一个 Contains 函数，这个 Contains 函数测试一个点是否位于矩形内部。这个函数使用了另一个 Windows API 函数来进行测试：

```
BOOL rectangle::Contains(const point & Point)const
```

```
{
return PtInRect(this,Point);
}
```

两个矩形对象相加的赋值操作符扩展了原来的矩形，成为两个矩形的并。

```
rectangle & rectangle::operator+=(const rectangle & Rect)
{
UnionRect(this,this,&Rect);
return *this;
}
```

rectangle 对象加上一个 size 对象的结果是对这个矩形的扩大，它也调用了一个 Windows API 函数：

```
rectangle & rectangle::operator+=(const size & Size)
{
inflateRect(this,Size.cx,Size.cy);
return *this;
}
```

另一方面，并没有叫做“InflateRect”的 API 函数，所以我们必须自己来写一个。其主要思想是不允许一个矩形为负。因此，如果你从一个矩形中减去一个比它还大的大小，这个矩形宽或高就会为零：

```
rectangle & rectangle::operator-=(const size & Size)
{
right-=Size.cx;
bottom-=Size.cy;
if(right<left)right=left;
if(bottom<top)bottom=top;
return *this;
}
```

4.7 处理彩色

PORTAGDI 库的 color 类实际上或多或少地是从我在《Borland C++内幕》中设计的 TXColor）以及相关的类）借用的。但是那些类是从 ObjectWindows 的 TColor 类派生出来的，而 color 类为了适于类库使用，并没有设计成派生类。color 定义在文件 COLOR. CPP 中。

你知道，Windows 的颜色是用 COLORREF 值来存贮的。COLORREF 是一个四字节的值，其中前三个字节分别存贮 R(红)、G(绿)、B(蓝)的值。最后一个字节没有用。

在设计 color 类时，我希望程序员能以下面的方式使用颜色：

```
color MyColor(color::Red);
MyColor+=green(64);
MyColor/=2;
```

第一个语句说明和定义了 MyColor，并初始化为红色。第二个语句将绿基色增加了一个值

（但不影响红和蓝基色值）；第三个语句导致整个颜色都暗了，因为亮度小了一半。

为了能实现上述思想，在我们真正创建 color 类前，我们需要创建一个 colorbyte 类。这个类要作为 red，green，blue 类的基类。下面是它的说明：

```
class colorbyte
{
public;
colorbyte(BYTE aValue): Value(aValue){}
colorbyte(colorbyte &. aColorbyte): Value(aColorbyte. Value){}
colorbyte &. operator=(BYTE aValue){Value=aValue;return *this;}
colorbyte &. operator=(colorbyte &. aColorbyte){Value=aColorbyte. Value;return *this;}
colorbyte &. operator+=(int aValue);
colorbyte &. operator+=(Colorbyte &. aColorbyte){retum *this+=aColorbyte. Value;}
colorbyte &. operator-=(int aValue);
colorbyte &. operator-=(Colorbyte &. aColorbyte){return *this-=aColorbyte. Value}
colorbyte &. operator*=(int aValue);
colorbyte &. operator/=(int aValue);
colorbyte operator+(int aValue)const{return colorbyte(*this)+=aValue;}
colorbyte operator+(colorbyte aColorbyte)const{return colorbyte(*this)+=aColorbyte}
colorbyte operator-(int aValue)const{return colorbyte(*this)-aValue}
colorbyte operator-(colorbyte aColorbyte)const{return colorbyte(*this-=aColorbyte}
colorbyte operator*(int aValue)const {return colorbyte(*this)*=aValue}
colorbyte operator/(int aValue)const{return colorbyte(*this)/=aValue}
operator BYTE()cosnt} {return Value}
Protected;
BYTE Value;
};
```

一个 colorbyte 对象与常规的 unsigned chor(无符号字节)变量唯一的不同是当算术运算使一个 colorbyte 对象的值超过255或小于0时并不回绕，而是保持最小或最大值。这个工作由 Normalize()来完成，这个函数定义在 COLOR。CPP 中：

```
static BYTE near Normalize(int Test)
{
if(Test>255)return 255;
else if(Test<0)return 0;
else return (BYTE)Test;
}
```

这个函数使 colorbyte 类实现所有的算术操作符成为可能。下面是算术赋值操作符的实现：

```
colorbyte &. colorbyte::operator+=(int aValue)
{
Value=Normalize((int)Value+aValue);
```

```
return *this;
}
colorbyte &colorbyte::operator=(int aValue)
{
Value-Normalize((int)Value-aValue);
return *this;
}
colorbyte &colorbyte::operator*=(int aValue)
{
Value=Normalize((int)Value*aValue);
return *this;
}
colorbyte &colorbyte::operator/-(int aValue)
{
Value=Normalize((int)Value/aValue);
return *this;
}
```

在类的说明中我们可以看到，非赋值算术操作等被实现为内嵌(**inline**)型函数。它们只是简单地构造一个中间对象，并对这个对象进行适当的算术运算。

4.7.1 创建基色类

我们下一步的任务是从colorbyte类派生出基色类。我们将需要用类来表示RGB三基色的值。除了名字外，这些类是相同的。我们可以使用C++的模板功能来实现它们。但不幸的是，并非所有的编译器都支持模板。Microsoft Visual C++1.5x就不支持模板，Visual C++2.0虽然支持模板，但它不能生成16位代码。

所以我们只好使用#define语句代替模板：

```
#define DECLARE_COLORBYTE_CLASS(COLORNAME)\
class COLORNAME: public colorbyte\
{\
public:\
COLORNAME(COLORNAME & V): colorbyte(V.Value){}\
COLORNAME(BYTE aValue=0): colorbyte(aValue){}\
COLORNAME & operator=(BYTE aValue)\
{return(COLORNAME &)((colorbyte &)*this=aValue);} \
COLORNAME & operator=(COLORNAME & V)\
{return((COLORNAME &)((colorbyte &)*this=V));} \
COLORNAME & operator+=(int aValue)\
{return((COLORNAME &)((colorbyte &)*this-V));} \
COLORNAME & operator+=(COLORNAME & V)\
{return((COLORNAME &)((colorbyte &)*this+=V));} \
```

```
COLORNAME & operator-=(int aValue)\
{return((COLORNAME &)((colorbyte &)*this-=aValue));} \
COLORNAME & operator-=(COLORNAME & V)\
{return(COLORNAME &)((colorbyte &)*this+=V));} \
COLORNAME & operator*=(int aValue)\
{return(COLORNAME &)((colorbyte &)*this*-aValue));} \
COLORNAME & operator/=(int aValue)\
{return((COLORNAME &)((colorbyte &)*this/=aValue));} \
} \
DECLARE_COLORBYTE_CLASS(red);
DECLARE_COLORBYTE_CLASS(green);
DECLARE_COLORBYTE_CLASS(blue);
```

既然我们说明和定义了 colorbyte 派生类(它所有的方法都是内嵌的)，我们现在可以定义 color 类了：

```
class color
  {
  public:
     color()
        : RValue(0), GValue(0), BValue(0){}
     color(red R,green G,blue B)
        : RValue(R), GValue(G), BValue(B){}
     color(COLORREF aValue){*this=aValue;}
     enum systemcolor
     {
     Scrollbar=COLOR_SCROLLBAR,
     Background=COLOR_BACKGROUND,
     ActiveCaption=COLOR_ACTIVECAPTION,
     InactiveCaption=COLOR_INACTIVECAPTION,
     Menu=COLOR_MENU,
     Window=COLOR_WINDOW,
     WindowFrame=COLOR_WINDOWFRAME,
     MenuText=COLOR_MENUTEXT,
     WindowText=COLOR_WINDOWTEXT,
     CaptionText=COLOR_CAPTIONTEXT,
     ActiveBorder=COLOR_ACTIVEBORDER,
     InactiveBorder=COLOR_INACTIVEBORDER,
     AppWorkspace=COLOR_APPWORKSPACE,
     Highlight=COLOR_HIGHLIGHT,
     HighlightText=COLOR_HIGHLIGHTTEXT,
     ButtonFace=COLOR_BTNFACE,
```

```
ButtonShadow=COLOR_BTNSIIADOW,
GrayText=COLOR_GRAYTEXT,
ButtonText=COLOR_BTNTEXT,
InactiveCaptionText=COLOR_INACTIVECAPTIONTEXT,
ButtonHighlight=COLOR_BTNHIGHLIGHT,
//These remaining colors are used in Windows 95 but not defined
//in the 16-bit WINDOWS. H
ButtonDarkShadow=21,//COLOR_3DDKSHADOW
ButtonLight=22,//COLOR_3DLIGHT
ToolTipText=23,//COLOR_INFOTEXT
ToolTip=24//COLOR_INFOBK
}
color(systemcolor anIndex){*this=anIndex;}
color & operaior-(COLORREF aValue);
color & operator=(systemcolor anIndex);
color & operator=(red R);
color & operator=(green G);
color & operator=(blue B);
short operator==(const color & aColor)const;
short operator==(COLORREF aValue)const;
    {return(*this==color(aValue));}
color & operator+-(color aColor);
color & operator+=(red R);
color & operator+=(green G);
color & operator+=(blue B);
color operator+(color aColor)const
    {return color(*this)+=aColor;}
color operator+(red R)const
    {return color(*this)+=R;}
color operator+(green G)const
    {return color(*this)+=G;}
color operator+(blue B)const
    {return color(*this)+=B;}
color & operator-=(color aColor);
color & operator-=(red R);
color & operator-=(green G);
color & operator-=(blue B);
color operator-(color aColor)const
    {return color(*this)-=aColor;}
color operator-(red R)const
```

```
        {return color(*this)-=R;}
    color operator-(green G)const
        {return color(*this)-=G;}
    color operator-(blue B)const
        {return color(*this)-=B;}
    color & operator*=(int aValue);
    color operator*(int aValue)const
        {return color(*this)*=aValue;}
    color & operator/=(int aValue);
    color oprtsior/(int aValue)const
        {return color(*this)/=aValue;}
    color & operator>>=(int Shift);
    color operator>> (int Shift) const
        {return color(*this)>>=Shift;}
    color & operator<<=(int Shift);
    color operator<<(int Shift)const
        {return color(*this)<<=Shift;}
    operator COLORREF()const
        {return RGB(RValue,GValue,BValue);}
    operator red()const
        {return RValue;}
    operator green()const
        {return GValue;}
    operator blue()const
        {return BValue;}
    static const COLORREF Red;
    static const COLORREF Green;
    static const COLORREF Blue;
    static const COLORREF Yellow;
    static const COLORREF Violet;
    static const COLORREF White;
    static const COLORREF Black;
private:
    red RValue;
    green GValue;
    blue BValue;
};
```

首先看看那些内嵌(**inline**)函数。你会发现那些非常简单的函数，包括大部分构造函数和非赋值操作等，都被定义成内嵌型。我用这种方式来降低函数头的代码空间。

你会注意到有两个构造函数(其中的一个用 COLORREF 来构造 color 对象，而另一个用

systemcolor 来构造 color 对象)是使用赋值操作符来实现的。用 COLORREF 来构造 color 对象，我们应该调用 Windows API 宏来实现：

```
color & color::operator=(COLORREF Value)
{
RValue=GetRValue(Value);
GValue=GetGValue(Value);
BValue=GetBValue(Value);
return *this;
}
```

用一个系统颜色值来构造 color 对象则使用另一种完全不同的方法。系统颜色实际上是一组用户设置的 Windows GUI 中各屏幕元素(如菜单、菜单文本等等)的颜色。Windows API 有一个函数 GetSysColor()可以获取这些颜色。当然，你有可能向这个函数传递一个非法的颜色索引。这里系统颜色被定义成枚举型变量，并确保没有非法颜色索引。因此，在这里不会向 GetSysColor()函数传递非法参数：

```
color & color::operator=(systemcolor anIndex)
{
return *this=GetSysColor(anIndex);
}
```

我们可以这样使用这个构造函数：

```
color MyColor(color::Window);
```

每个只接收一个 colorbyte 类型的值的赋值操作符自动将该值传递给对应的数据成员，比如：

```
color & color::operator=(red R)
{
RValue=R;
return *this;
}
```

这就允许程序员单独对某个基色赋值：

```
color MyColor;
MyColor=green(128);
MyColor=blue(64);
```

等价测试是这样的：

```
short color::operator==(const color & aColor)const
{
return((RValue==aColor.RValue)&&(GValue==aColor.GValue)
&&(Bvalue==aColor.BValue));
}
```

你可能想增加每个基色的等价操作符，我也想这样做，但还是觉得这样的代码很容易让人糊涂。

如果你把一个颜色加上另一个颜色，每个基色将分别相加：

```
color & color::operator+=(color aColor)
{
RValue+=aColor.RValue;
GValue+=aColor.GValue;
BValue+=aColor.BValue;
return *this;
}
```

这就允许你将相同的红色和绿色相加而得到黄色：

```
color MyColor(color::Red);
color Green(color::green);
MyColor+=Green; //现在 MyColor 表示黄色了
```

但单独加一种颜色只会影响相应的基色值，colorbyte 类会自动处理值的溢出：

```
color & color::operator+=(red R)
{
RVaiue+=R;
return *this;
}
```

与此相似，乘法和除法操作符被设计成颜色强度的改变，这种改变被分别传递给三种基色：

```
color & color::operator*=(int aValue)
{
RValue*=aValue;
GValue*=aValue;
BValue*=aValue;
return *this;
}
```

我得考虑一下向右和向左移位操作符到底是什么意思。我猜想你一定会与我争论这个问题，但我最后决定将基色调换位置，比如：向右移位就是将绿变红、蓝变绿、红变蓝：

```
color & color::operator>>=(int Shift)
{
Shift%=3;
while(Shift>0)
{
colorbyte Temp(BValue);
BValue=GValue;
GValue=RValue;
RValue=Temp;
  }
return *this;
}
```

向左移位的操作也是相同的，只是方向不同。在实现类的文件中还有一些预定义颜色常量的定义：

```
const color color::Red=RGB(255,0,0);
const color color::Green=RGB(0,255,0);
const color color::Blue=RGB(0,0,255);
const color color::Yellow=RGB(255,255,0);
const color color::Violet=RGB(255,0,255);
const color color::White=RGB(255,255,255);
const color color::Black=RGB(0,0,0);
```

4.8 定义刷子类

最后，我们来实现一个真正的 GDI 对象：刷子。brush 类是 BRUSH. CPP 文件中所实现的第一个类。应用程序的设备环境中都有一个刷子。默认的是白色刷子。你可以用自己设计的刷子来代替它。下面是 brush 类的说明：

```
class brush
    {
    friend class devicecontext;
    public:
        enum stockbrushes
            {
            White=WHITE_BRUSH,
            LightGray=LTGRAY_BRUSH,
            Gray=GRAY_BRUSH,
            DarkGray=DKGRAY_BRUSH,
            Black=BLACK_BRUSH;
            Invisible=HOLLOW BRUSH
            }
        brush(stockbrushes StockBrush);
        enum hatchstyles
            {
            Horizontal=HS_HORIZONTAL,
            Vertical=HS_VERTICAL,
            ForewardDiagonal=HS_FDIAGONAL,
            BackwardDiagonal=HS_BDIAGONAL,
            Cross=HS_CROSS,
            DiagonalCross=HS_DIAGCROSS
            }
        brush(color aColor,hatchstyles style);
        brush(HBITMAP aBitmap);
```

```
    brush(color aColor);
    ~brush ();
    operator HBRUSH(void){return Handle;}
private:
    BOOL StockObject;
    HBRUSH Handle;
    devicecontext* pDC;
};
```

与所有其它的 GDI 对象一样，Windows API 中刷子是用句柄来表示的(具体地说，是 HBRUSH)。你可以设计一个从 HBRUSH 构造 brush 对象的构造函数，但我认为那是鼓励“半面向对象”的程序设计。而我则倾向于定义向 HBRUSH 的类型强制转换，以便可以在使用 HBRUSH 的任何地方使用 brush 对象。

获取刷子的一种方法是从 Windows 的仓库中借取。枚举型变量 stockbrushes 提供了各种库存刷子的标识符。标识符的值与 Windows API 中定义的非常类似，但利用枚举型变量我们可以使 C++编译器避免程序员偶尔会以不适当的值(比如说，*BLACK_PEN*)创建一个刷子。我们可以调用 Windows API 函数 GetStockObject()来从 Windows 库存对象中获取一个刷子。由于这个函数返回的是一般的句柄，所以它的返回值必须经过类型强制转换：

```
brush::brush(stockbrushes StockBrush)
: Handle((HBRUSH)GetStockObject(StockBrush)),
StockObject(TRUE),pDC(NULL)
{
}
```

另一种刷子的类型是阴影刷子。用这种刷子刷过的区域填充的是彩色线而不是实心彩色。阴影有几种风格，枚举型变量 hatchstyles 提供了相应的标识符。图4.1显示了可用的阴影风格。

这里必须调用 Windows API 函数(GetHatchBrush()。但要注意参数传递的顺序与构造函数相反。我认为程序员更愿意接受把刷子的颜色做为第一个参数的做法：

```
brush::brush(color aColor hatchstyles Style)
: Handle(CreateHatchBrush(Style,aColor)),
StockObject(FALSE), pDC(NULL)
{
}
```

如果你有一个8X8点阵的位图，你可以用它创建一个刷子。这种刷子叫做“模板刷子”。这需要调用 Windows API 函数 CreatePatternBrush()：

```
brush::brush(HBITMAP aBitmap)
: Handle(CreatePatternBrush(aBitmap)),
StockObject(FALSE), pDC(NULL)
{
```

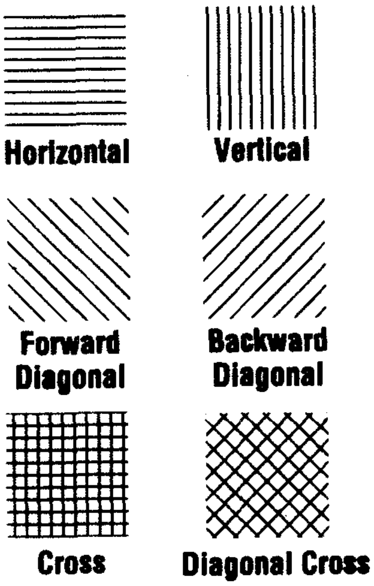

图4.1　六种阴影风格

```
}
```

顺便提一下，如果你的位图是单色的，刷子将使用设备环境中当前文本颜色和背景颜色来显示位图。并且如果位图大于8X8点阵，那么只使用左上角8X8的部分。

最后一个构造函数用于创建实心颜色刷子：

```
brush::brush(color aColor)
: Handle(CreateSolidBrush(aColor)), StockObject(FALSE), pDC(NULL){}
```

虽然这种刷子叫做实心刷子，但你也应知道并非所有的用户都在使用16M 颜色的显示驱动程序。所以并非所有的颜色都可以显示成真的实心颜色。在有些情况下，Windows 将自动提供一个抖动刷子。尽管你可能只是听说过"抖动"这个词，但这个词并不是当前的新发明。实际上，它是个旧的打印机术语，它的意思是使用各种基色的小点来显示非基色颜色。

如果你特别需要最接近的颜色，Windows 的驱动程序会显示哪种真的实心颜色呢?WindowsAPI 中有一个函数 GetNearestColor()，它以一个设备环境的句柄和 COLORREF 值做为参数，返回与 COLORREF 值最接近的实心颜色值：

```
color MyColor(233,21,28);
devicecontext MyDC(Window);
brush MyBrush(GetNearestColor(MyDC,MyColor));
```

创建的刷子必须要删除，但不要删除库存对象。这些是在析构函数中完成的：

```
brush::~brush()
{
if(pDC)pDC->RestoreBrush();
if(lStockObject)DeleteObject(Handle);
}
```

你可能还记得，在类的说明中，devicecontext 是一个友元类。有一点特别重要，就是当一个刷子被选择在设备环境中时，不能被删除。对 RestoreBrush()的调用就是为了避免这种情况的发生。(我们在后面会见到 devicecontext 类的说明和定义)

4.9 不要忘记你的画笔!

"画笔"对象是在 PEN. CPP 文件中实现的，它与刷子对象很类似。一个设备环境在生成时有一个默认的画笔。画笔可以从库存对象中获得，也可以被创建。

下面是 pen 类的说明：

```
class pen
    {
    friend class devicecontext;
    public:
        enum penstyle
            {
            Solid=PS_SOLID,
            Dashed=PS_DASH,
            Dotted=PS_DOT,
```

```
            DashDot=PS_DASHDOT,
            DashDotDot=PS_DASHDOTDOT,
            InsideFrame=PS_INSIDEFRAME
            };
        pen(color Color,int Width=0,penstyle Style=Solid);
        enum stockpens
            {
            White=WHITE_PEN,
            Black=BLACK_PEN,
            Invisible=NULL_PEN
            };
        pen(stockpens StockPen=Black);
        ~pen();
        operator HPEN(void){return Handle;}
    private;
        BOOL StockObject;
        HPEN Handie;
        devicecontext * pDC;
    };
```

GetStockObject()函数不但可以返回刷子，还可以返回画笔(也可以返回库存字体)。与刷子类的构造函数一样，pen类的构造函数也使用这个函数的返回值，默认的是黑画笔：

```
pen::pen(stockpens StockPen)
    : Handle((HPEN)GetStockObject(StockPen)), StockObject(TRUE), pDC(NULL)
{
}
```

画笔对象可以使用特定的颜色、宽度以及风格来创建。在Windows API中这三个参数必不可少：但由于pen类的构造函数可以使用参数的默认值，所以只需给出颜色值就可以了。这里调用了Windows API函数Createpen：

```
pen::pen(color Color,int Width,penstyle Style)
    : Handle(CreatePen(Style,Width,Color)), StockObject(TRUE), pDC(NULL)
{
}
```

与刷子类一样，如果pen对象不是库存对象，pen类的析构函数将删除这个pen类对象：

```
pen::~pen()
{
if(pDC)pDC->Restorepen();
if(!StockObject)DeleteObject(Handle);
}
```

4.10 定义设备环境类

devicecontext 类是整个 GDI 系统的基础。它定义在 DEVICE. CPP 中。设备环境是 Windows 维护设备无关性的手段。这真是创造性的一举。同样一组函数可以用于在任何一种设备上作图，根本不用考虑设备是打印机还是显示器。

这本来是好事。但对于 C 程序员来说，麻烦的是存在着几种不同种类的设备环境，他们得以不同的方式来创建和删除这些设备环境。一般的人谁会记住哪个是哪个呀！

可我们不是 C 程序员。我们可以通过将 GDI 功能放在类库中把那些差异隐藏起来，而在派生类中处理那些差异。

下面是基类 devicecontext 类的说明：

```
class devicecontext
    {
    public:
        devicecontext(HDC aHandle);
        virtual~devicecontext();
        operator const HDC(void){return Handle;}
        devicecontext &. operator=(brush &. Brush);
        devicecontext &. operator=(pen &. pen);
        devicecontext &. operator=(color Color);
        void RestoreBrush(void);
        void RestorePen(void);
        virtual rectangle GetWindowRect(void)const;
        virtual rectangle GetClientRect(void)const;
        virtual rectangle GetClipRect(void)const;
        void SetBackgroundColor(color Color);
        void SetTabStops (int aTabStops){TabStops=aTabStops;}
        enum drawmode
            {
            Transparent=TRANSPARENT,
            Opaque=OPAQUE
            }
        void SetBackgroundDrawMode (drawmode DrawMode);
        void DrawRectangle (rectangle Rectangle);
        void DrawEllipse (rectangle Rectangle);
        void DrawFocusRectangle (rectangle Rectangle);
        void DrawText (point Point,const char * Text);
        enum alignment
            {
            TopLeft=DT_LEFT,
```

```
        MiddleLeft=DT_LEFT|DT_VCENTER|DT_SINGLELINE,
        BottomLeft=DT_LEFT|DT_BOTTOM|DT_SINGLELINE,
        TopCenter=DT_CENTER,
        MiddleCenter=DT_CENTER|DT_VCENTER|DT_SINGLELINE,
        BottomCenter=DT_CENTER|DT_BOTTOM|DT_SINGLELINE,
        TopRight=DT_RIGHT,
        MiddieRight=DT_RIGHT|DT_VCENTER|DT_SINGLELINE,
        BottomRight=DT_RIGHT|DT_BOTTOM|DT_SINGLELINE
        };
    void Draw Text (rectangle Rectangle,
        const char * Text,alignment Alignment=MiddleCenter);
    void MoveTo (point Point){::MoveTo (Handle,Point.x,Point.y);}
    void LineTo (point Point){::LineTo (Handle,Point.x,Point.y);}
protected:
    HDC Handle;
    void Restore (void);
private:
    int TabStops;
    HBRUSH OldBrush;
    brush * CurrentBrush;
    HPEN OldPen;
    Pen * CurrentPen;
};
```

你可能会注意到我并没有试图在这个类中复制所有的作图功能。实际上，你用 Windows API 可以做的事情，在这里也可以做得到。但做为一个教学工具，许多方法隐藏了类的核心。我鼓励你增加你需要的方法，这很容易做到。这里只实现了 VBX 类库需要的所有功能。

这个基类的构造函数需要一个已有的设备环境的句柄：

```
devicecontext::devicecontext(HDC aHandle)
    : Handle(aHandle), TabStops(4), OldBrush(NULL), CurrentBrush(NULL),
    OldPen(NULL), CurrentPen(NULL)
{
}
```

析构函数什么都没做。它是个虚函数，在基类中这样设置是很适当的。即使你可能不需要各种 devicecontext 对象的容器，这种做法也是不错的：

```
devicecontext::~devicecontext()
{
}
```

你以前肯定觉得 SelectObject()这个函数非常奇怪，是不是?它可以把画笔、刷子、字体以及其它对象选入设备环境中，并且知道怎样处理它们。为什么我们要把这种疯狂的做法带进 GDI 类库中呢?在 devicecontext 类中，这种操作是使用赋值操作符来完成的。当操作符右边的

对象是 brush 或 pen 时，SelectObject()函数会被调用；但至少程序员不会意识到这个调用。

```
devicecontext & devicecontext::operator=(brush & Brush)
{
HBRUSH PreviousBrush=(HBRUSH)::SelectObject(Handle,Brush);
if(CurrentBrush)CurrentBrush->pDC=NULL;
if(!OldBrush)OldBrush=PreviousBrush;
CurrentBrush=&Brush;
Brush. pDC=this;
return *this;
}
devicecontext & devicecontext::operalor=(pen & Pen)
{
HPEN PreviousPen=(HPEN)::SelectObject(Handle,Pen);
if(CurrentPen)CurrentPen->pDC=NULL;
if(!OldPen)OldPen=PreviousPen;
CurrentPen=&Pen;
Pen. pDC=this;
return *this;
}
```

对 CurrentBrush 和 CurrentPen 这两个属性的判断是为了使当前刷子和画笔知道它们是否在使用。如果要在删除设备环境以前删除刷子或画笔的话，就要分别调用 RestoreBrush()、RestorePen()将它们选出设备环境，这时会对这两个属性进行操作：

```
void devicecontext::RestoreBrush(void)
{
if(OldBrush)
  {
  ::SelectObject(Handle,OldBrush);
  CurrentBrush->pDC=NULL;
  CurrentBrush=NULL;
  }
}
void devicecontext::RestorePen(void)
{
if(OldPen)
  {
  ::SelectObject(Handle,OldPen);
  CurrentPen->pDC=NULL;
  CurrentPen=NULL;
  }
}
```

当操作符右边的对象是 color 时，设备环境中的本文颜色将会改变：

```
devicecontext &. devicecontext::operator=(color Color)
{
::SetText Color(Handle,Color);
return *this;
}
```

我本来非常想用减法赋值操作符来设置背景颜色。但这种做法对于我来说甚至也是很勉强的。所以我使用了常规命名的方法。但我并没有使用 Microsoft 程序员所喜好的缩写形式来命名，而是起了一个有意义的名字：

```
inline void devicecontext::SetBackgroundColor(color Color)
{
::SetBkColor(Handle,Color);
}
inline void devicecontext::SetBackgroundDrawMode(drawmode DrawMode)
{
::SetBkMode(Handle,DrawMode);
}
```

GetWindowRect()、GetClientRect()、GetClipRect()这三个纯虚函数使得作图函数可以向设备环境询问局部信息而不需考虑设备环境的类型。

这里我给 Rectangle()，Ellipse()和 DrawFocusRectangle()这三个函数起了更清晰的名字：

```
void devicecontext::DrawRectangle(rectangle Rectangle)
{
::Rectangle(Handle,Rectangle. left,Rectangle. top,Rectangle. right,Rectangle. bottom);
}
void devicecontext::DrawEllipse(rectangle Rectangle)
{
::Ellipse(Handle,Rectangle. left Rectangle top,Rectangle. right,Rectangle. bottom);
}
void devicecontext::DrawFocusRectangle(rectangle Rectangle)
{
::DrawFocusRect(Handle,&.Rectangle);
}
```

Windows API 提供了多种方法将文本显示在设备环境中，如 TextOut() ExtTextOut()，TabbedTextOut()，DrawText()等等。在 devicecontext 类中，所有的作图函数名都以“Draw”开头，并且定义了很多重载的 DrawText()函数，它们根据不同的参数来实现文本显示功能。例如，如果你有一个点(point)对象和一个字符串，则可以使用下面的函数：

```
void devicecontext::DrawText(point Point,const char *Text)
{
::TextOut(Handle,Point. x,Point. y,Text,strlen(Text);
}
```

另外，如果你希望将文本以某种对齐方式放在一个矩形中，你可以使用下面的函数：

```
void devicecontext :: DrawText ( rectangle Rectangle, const char * Text, alignment
Alignment)
{
::DrawText(Handle,Text,-1,&Rectangle,
  (Alignment|DT_EXPANDTABS|DT_TABSTOP)+(TabStops<<8));
}
```

我们要了解的最后一个方法是 Restore()。这个方法是保护的(**protected**)，并且它只是调用了 RestoreBrush()和 RestorePen()。这个方法将在派生类中使用以确保在返回设备环境前恢复原来的 GDI 对象：

```
void devicecontext::Restore(void)
{
RestoreBrush();
RestorePen();
}
```

4.10.1 创建作图环境(paintcontext)类

windows 中大多数作图操作都发生在对 WM_PAINT 消息的响应中；程序员将通过调用 Windows API 函数 BeginPaint()来获取设备环境的句柄。在我们这个小型 GDI 类库中，表示这种设备环境的类是 Paintcontext 类：

```
class paintcontext::public devicecontext
    {
    public:
        paintcontext(HWND aWindow);
        ~paintcontext();
        paintcontext & operator=(brush & Brush);
        paintcontext & operator=(pen & Pen);
        BOOL ShouldRedrawBackground (void) const{return ps. fErase;}
        virtuai rectangle GetClipRect (void) const;
    private:
        HWND Window;
        PAINTSTRUCT ps;
    };
```

BeginPaint()函数是在构造函数中调用的，而 EndPaint()则是在析构函数中，当设备环境被恢复成原来状态后调用的：

```
paintcontext::paintcontext (HWND aWindow): devicecontext (BeginPaint (aWindow,
&ps)),
        Window (aWindow)
    {
    }
```

```
paintcontext::~paintcontext()
    {
    Restore ();
    EndPaint (Window,&ps);
    Handle=NULL;
    }
```

类中必须定义赋值操作符，因为赋值操作符不会被继承。但在头文件中的这一对内嵌式操作符函数是使用了类型强制转换后，调用基类中的函数来实现的：

```
inline paintcontext & paintcontext::operator=(brush & Brush)
    {
    return (paintcontext &)(((devicecontext &)*this)=Brush);
    }
inline paintcontext & paintcontext::operator=(pen & Pen)
    {
    return (paintcontext &)(((devicecontext &) *this)=Pen);
    }
```

GetWindowRect()的实现只是简单地调用同名的 Windows API 函数，再加入一些代码以返回一个 rectangle 对象：

```
rectangle clientcontext::GetWindowRect (void) const
    {
    RECT Rect;
    ::GetWindowRect (Window,&Rect);
    return rectangle (Rcet);
    }
```

GetClientRect()也把调用传递给 API 函数。但 GetClipRect()函数则使用 PAINTSTRUCT 结构类型的成员数据 ps，返回相应的值：

```
rectangle paintcontext::GetClipRect(void)const
    {
    return rectangle (ps.rcPaint);
    }
```

4.10.2 创建用户区域设备环境(clientcontext)类

除了响应 WM_PAINT 消息，有时你还需要在其它情况下作图，比如在响应鼠标按键或键盘击键时。这种情况下，就需要创建一个 clientcontext 类对象：

```
class clientcontext: public devicecontext
    {
    public:
        clientcontext (HWND aWindow);
        ~clientcontext ();
        clientcontext & operator=(brush & Brush);
```

```
        clientcontext & operator=(pen & Pen);
        virtual rectangle GetWindowRect (void) const;
        virtual rectangle GetClientRect (void) const;
        virtual rectangle GetClipRect (void) const;
    private:
        HWND Window;
    };
```

它与 paintcontext 类唯一不同的地方是它的构造函数。它的构造函数必须调用 GetDC()，而析构函数必须调用 ReleaseDC()。

```
clientcontext::clientcontext (HWND aWindow)
        devicecontext (GetDC (aWindow)),
        Window (aWindow)
    {
    }
clientcontext::~clientcontext ()
    {
    Restore ();
    ReleaseDC (Window,Handle);
    Handle=NULL;
    }
```

到此为止，全部 GDI 类库都定义完毕了。当然你可以编写一段小程序测试一下它们的有效性。

第五章　创建一个与软件商无关的 text 类

我不知道你，但我却知道 Microsoft 公司和 Borland 公司之间的争斗。Borland 公司都打红眼了，竟然将十倍于 Microsoft 公司的同类产品性能的 Pasoal 编译器的价钱降到人家的十分之一还少。今天 Microsoft Visual C++和 Borland C++仍在竞争，丝毫没有和谈的迹象。可是，倒霉的是谁呢?是我们这些作为旁观者的程序员。我们用 MFC 编写的应用程序不能在 Borland C++下编译，而用 ObjectWindows 编写的程序又不能在 Visual C++下编译。虽然每种产品都声称兼容 ANSI 标准，但事实上 ANSI 标准并非那么有力，并且这两个公司都在以他们自己的方式完善这个标准。

当然，我们可以得过且过。但我不是愿意挨到哪家公司最后取得胜利的那种人。我决定以我的方式进行抗击。抗击方法之一就是创建你自己的、不依赖于任何一家公司的类库。你拥有自己的类库越多，你对那些所谓的软件工业的英雄们的依赖就越小。

乍看起来，这种做法对你来说好象是不可能的。但如果你真的使用过面向对象的风格编写过各种类，你就会知道这种做法并非不可能。上一章我们讲述了如何创建与软件商无关的 Windows GDI 类库。现在我们进一步放开手脚，来设计和实现一个真正实用的处理文本的类。

5.1　创建 text 类

ANSI C++标准委员会已经定义了一个字符串类(string 类)。有人认为它是字符串类的最后标准。但不幸的是，标准委员会仍然偏爱 UNIX，所以他们的 String 类并不支持 Windows 格式的字符串资源(尽管它在 TTY 控制台上工作得很好)。为了应付这种情况，Borland 和 Microsoft 公司实现了它们自己的字符串类:Microsoft 公司的 CString 和 Borland 公司的 string。其中 Borland 公司的 string 类比较接近 ANSI 标准。

C 程序员或许觉得很奇怪，但事实上一个“字符串”确实应该是一个以 NULL 结尾的字符数组。其它语言也都是这么处理字符串的。C 语言成为例外的唯一原因就是 C 标准库中处理以 NULL 结尾的字符串的库函数少得可怜。

这三种字符串类都提供了处理子串、删除串的一部分、查找特定字符等等方法。我们也想这样做，只不过是用更自然的 C++的方式。而且由于我们只对包含有文本的字符串子集感兴趣，所以我们把我们的新类叫做 text(以避免又增加一个“string”名词)并且只实现了那些我们现在需要的功能。

5.1.1　定义头文件

通常我们在分析一个类时，往往是从头文件中类的说明开始的。这里我们先从 TEXT. H 文件开始。这个文件以 # **ifndef** 编译宏结构开始以避免过多地包含这个头文件(前一章已讲过了)。文件中还有几个包含语句，这是因为在文件中定义的内嵌(**inline**)函数要用到这些库头文件说明中的函数。

```
#ifndef_TEXT_H_
#define_TEXT_H_
```

```
#include<windows.h>
#include<stdlib.h>
#include<math.h>
```

下面是辅助类 range 类。这个类在后面会用得到，它使从 text 对象中抽取子串的操作变得很容易。这个类完全在头文件中实现了：

```
class range
{
public:
  range (int aFirst,int aLast)
       : First(aFirst)、Last(aLast){}
private:
    int First, Last;
friend class text;
};
```

我把它放在这个位置上是为了使你能很容易找到它。如何使用和为什么使用 range 类很快就会清楚的。

同样，下一个类也是辅助类。它使你在一个单独的对象中既可以使用字符串表 ID 也可以使用含有那个字符串表的模块实例。(记住，在 Windows 里，字符串不但可以从当前正装入的模块中，而且也可以从已装入的模块中装入。)下面是它的说明：

```
class stringID
{
public:
    stringID(UINT anID,HINSTANCE anInstance=NULL)
         :    ID(anID),
              Instance(anInstance){}
private:
    UINT ID;
    HINSTANCE Instance;
friend class text;
};
```

再后面才是 text 类的说明。

通常，我呈现全部的 class 语句并且在实现文件中定义类的所有的成员函数。但是，text 类中定义了太多的内嵌函数，上述做法是不实用的。所以，这里我把类的函数分成组来说明和实现。如果你觉得这样做不大好懂，你可以将头文件和实现(定义)文件调入你喜爱的编辑器中，这样就可以看到一组一组的实际代码了。

5.1.2 text 类构造函数和析构函数

和以前一样，class 语句以类的构造函数和析构函数的说明开始：

```
class text
```

```
public:
    text(const char*Original=0,BOOL aCaseSensitive=FALSE);
    text(char Original,short Count=1,BOOL aCaseSensitive=FALSE);
    text(const text & Original);
    text(long Value);
    text(double Value);
    text(stringID StringID,BOOL aCaseSensitive=FALSE);
    virtual~text();
```

⇓

⇓

你会注意到，我们可以用标准 C 风格的字符串、单个字符、该类对象的拷贝、长整数(**long**)以及双精度浮点数(**double**)来构造 text 类的对象。析构函数是虚函数。通常析构函数都应该设计成虚的，这使从这个类派生一个新类变得很容易。

text 类构造函数的实现代码放在了 TEXT. CPP 中：

```
#include "text. h"
#include<string. h>
#include<ctype. h>

text::text(const char*Original,BOOL aCaseSensitive)
:      Length(0), Storage(NULL), CaseSensitive(aCaseSensitive)
{
Store(Original);
}

text::text(char Original,short aCount,BOOL aCaseSensitive)
:      Length(0), Storage(NULL), CaseSensitive(aCaseSensitive)
{
store(Original,aCount);
}

text::text(const text & Original)
: Length(0), Storage(NULL), CaseSensitive(Original CaseSensitive)
{
Store(Original. Storage);
}

text::text(long Value)
:      Length(0), Storage(NULL), CaseSensitive(FALSE)
{
```

```
Store(Value);
}

text::text(double Value)
:       Length(0), Storage(NULL), CaseSensitive(FALSE)
{
Store(Value);
}

text::text(stringID StringID,BOOL aCaseSensitive)
:       Length(0), Storage(NULL), CaseSensitive(aCaseSensitive)
{
Store(StringID);
}
```

通过浏览这些构造函数你可以知道两件事情。第一件是类中有三个属性，它们是 Length、Storage 和 CaseSensitive。按惯例它们被说明成私有成员：

```
class text
{
        ⇓
        ⇓
private:
        char * Storage;
        int Length;
        BOOL CaseSensitive;
}
```

第二件是属性的实际赋值都由方法 Store()来完成。存在多个 Store()函数的重载版本，几乎每一个构造函数都对应一个 Store()函数。并且在调用 Store()之前 Storage 被初始化成 NULL。这使析构函数的设计变得简单：

```
text::~text()
{
delete[]Storage;
}
```

由于 delete 操作对 NULL 指针是安全的，所以没有必要对 Storage 进行复杂的检测，看它是否是有效指针。析构函数可以利索地将那些东西清除掉。

5.1.3 text 类的数据存贮

犹如秘密身份，Store()函数是保护的，不能在 text 类或 text 的派生类外部调用：

```
class text
{
        ⇓
```

⇓

```
protected:
    void Clear(void);
    void Store(const char * Original);
    void Store(char Original,short Count=1);
    void Store(long Value);
    void Store(double Value);
    void Store(stringID StringID);
    ⇓
    ⇓
```

Clear()函数也是保护的(protected)。各个 store()函数调用该函数用于将 text 对象以前的内容删除掉。这样，store()函数不仅可用于构造函数，也可以用于赋值。这就是为什么我们将 Storage 初始化为 NULL 的原因：

```
void text::Clear(void)
{
Length=0;
delete[]Storage;
Storage=NULL;
}
```

为了在清除旧的内容后将标准 C 风格的字符串存贮到 text 对象中，必须分配足够的内存空间以容纳字符串和结尾的 NULL 字符。这样标准 C 库函数可以用于将字符串复制到存贮区：

```
void text::Store(const char * Original)
{
Clear();
Length=Original?strlen(Original): 0
Storage=new char[Length+1];
if(Length)
    strcpy(Storage,Original);
else
    Storage[0]—0;
}
```

注意：上面的代码处理了以 NULL 指针调用构造函数(或后面的赋值操作符)的情况。在这种情况下，Store()函数存贮长度为0的字符串：即单独一个 NULL 结尾符，这样它就可以被那些希望使用既定缓冲区的函数安全地使用了。这个版本的 Store()函数被以 C 风格的字符串和对象拷贝做为参数的构造函数所调用。

存贮一个单字符或重复字符的操作稍有不同，好在 C 库函数 memset()为我们做了大部分的工作：

```
void text::Store(char Original,short Count)
```

```
{
Clear();
Length=Count;
Storage=new char[Length+1];
memset(Storage,Original,Count);
Storage[Length]=0;
}
```

有两个版本的 Store()函数分别接受 long 和 double 类型的参数。它们用于以二进制数来构造 text 对象。这里，我们又使用了 C 标准库函数：

```
void text::Store(long Value)
{
char Digits[48];
Itoa(Value,Digits,10);
Store(Digits);
}

void text::Store(double Value)
{
char Digits[48];
gcvt(Value,48,Digits);
Store(Digits);
}
```

注：这里我们也使用了自己的代码：在 itoa()或 gcvt()将二进制数值转换成字符串后调用以 C 字符串为参数的 Store()函数。

最后我们定义了以字符串 ID 为参数的重载函数 store()：

```
void text::Store(stringID StringID)
{
char Buffer[256];
LoadString(StringID,Instance,StringID. ID,
    Buffer,sizeof Buffer);
Store(Buffer);
}
```

LoadString()当然是 Windows API 函数，它可以从字符串表资源中获取字符串。由于字符串表的表项长度不会超过255个字符，所以256个字符长度的缓冲区足以容纳获取的字符串。最后调用标准 Store()函数将字符串放到数据堆中。

5.1.4 text 类的操作符函数

class 语句下面的部分是赋值操作符的定义：

```
class text
{
```

```
    ⇓
    ⇓
public:
    text & operator=(const text & Original);
    text & operator=(const char * Original)
        {Store(Original); return *this;}
    text & operator=(const char Original)
        {Store(Original); return *this;}
    text & operator=(long Value)
        {Store(Value); return *this;}
    text & operator=(double Value)
        {Store(Value); return *this;}
    text & operator=(stringID StringID)
        {Store(StringID); return *this;}

void SetCaseSensitive(BOOL aCaseSensitive=TRUE)
    {CaseSensitive=aCaseSensitive;}
BOOL IsCaseSensitive(void)const
     {return CaseSensitive;}
⇓
⇓
```

我在这部分中包含了设置和获取 CaseSensitive 属性值的监视函数。

当你看到我们有六个赋值操作符时，你不应感到惊奇，因为每一个操作符都针对着一个构造函数。这些赋值操作符应这样定义，它们使得当一个程序员使用这个类时，他不应再考虑可以将什么数据类型赋值给一个 text 对象。如果你想初始化一个 text 对象，你可以给这个对象赋值。

只有第一个操作符没有被定义成内嵌函数。其它的只是调用 Store()函数并返回其本身。为了能允许连续赋值，赋值操作符通常返回对对象本身的引用。如：a=b=c。那么第一个操作符是如何处理这种特殊情况的呢?很容易，它必须检查是否是在对本身进行赋值：

```
text & text::operator=(const text & Original)
{
if(this != & Original)
    Store(Original. Storage);
return *this;
}
```

如果你再检查一遍 Store()的代码，你会发现这个函数先删除了接收对象的内容，而后再从发送对象复制相应的值。如果两个对象是同一个对象，结果就是赋值一个空字符串。

你可能会问，谁又会把对象赋值给它本身呢!我也承认这不是一个有用的操作符。这就好象是让当官的向传令兵保守秘密一样。但 C++语言允许这样的语句：a=a，所以我们必须考虑这种情况。

5.1.5 谁会更快，是闪电还是超人?

在赋值操作符的后面是一个对文本字符串最一般的操作：比较。在C中，我们需要调用函数 strcmp()或它的47个相关函数中的某一个。当然，在这里我们仍需要这么做，因为实在没有必要再重写比较的算法。但我们至少可以将它隐藏起来，使它对程序员来说是不可见的。

其中的关键就是 Compare()这个保护(**Protected**)的函数：

```
class text
{
        ⇓
        ⇓
protected:
        int Compare(const text &. Test)const;
        ⇓
        ⇓
```

你已经见过 CaseSensitive 这个属性了。意思是大多数文本操作都是大小写不敏感(大小写无关)的。但仍有些操作是大小写敏感的。我们需要同时考虑这两种情况。因此，当你构造一个 text 对象时，你可以规定这个对象是否大小写敏感(默认为大小写不敏感)。

如果比较的双方中任何一个对象是大小写敏感的，那么比较操作就是大小写敏感的：

```
int text::Compare(const text &. Test)const
{
if(CaseSensitive||Test.CaseSensitive)
    return strcmp(Storage,Test.Storage);
else
    return stricmp(Storage,Test.Storage);
}
```

我们在 text 类中定义了许多比较操作符，而上面这个函数则满足了所有的比较操作符的需要。这里呈现的不仅是各种比较操作符，而且还包含有 text 类所支持的各种数据类型的比较操作符的重载函数。(但去掉了 **long** 和 **double** 这两个类型,因为 text 是文本类而不是数字类)。例如，下面这些是等价操作符的说明：

```
class text
{
        ⇓
        ⇓

public:
    short operator==(const text &. Test)const
          {return Compare(Test)==0;}
    short operator==(const char * Test)const
          {return *this==text(Test);}
    short operator==(char Test)const
```

```
        {return *this==text(Test);}
    short operator==(stringID Test)const
        {return (*this)==text(Test);}
    friend short operator==(const char * Test1,const text & Test2);
    friend short operator==(char Test1,const text & test2);
    friend short operator==(stringID Test1,const text & Test2);
    ⇓
    ⇓
```

其中，只有第一个真正调用了 Compare()函数，而其它只是创建一个暂时的 text 对象并调用第一个操作符函数。这意味着只有其中的一个操作符函数可能会含有错误(**bug**)，这种方法节省了调试时间。

那三个友元(**friend**)函数用来处理比较操作符的左边不是 **text** 类型的情况。它们也被定义成内嵌(**inline**)函数，但它们的代码必须在 **class** 语句外部编写，以便使它们位于头文件的底部。它们所要做的就是交换参数的位置。例如：

```
inline short operator==(const char * Test1,const text & Test2)
{
return(Test2==Test1);
}
```

我有个学生，他曾经强烈建议一个类不应简单地允许这种测试。为什么不把这些工作留给程序员按照类所支持的方式去编写呢？显然，那个学生只注意到了实现一个完整的对象类所需要的那些讨厌的代码。但面向对象程序设计的一个重要思想是构造新的类，并使它适合于某种语言就好象它是那种语言的一部分一样。此外，哪一位有强烈自尊心的超级英雄会考虑避开解决这种问题呢！

所有这六个比较操作符都被使用了，但我不想再为它们浪费篇幅，因为除了操作符本身外，代码都是一样的。

但我要指出的是只有等价和不等是可交换的，只有它们才允许 Test1和 Test2相互调换位置而无需改变操作符。为了你的方便，表5.1列出了一组比较操作。

表5.1　　一组比较操作

Test1==Test2	Test2==Test1
Test1<Test2	Test2>=Test1
Test1<=Test2	Test2>Test1
Test1>Test2	Test2<=Test1
Test1>=Test2	Test2<Test1
Test1!=Test2	Test2!=Test1

5.1.6 英雄的助手

就象超级英雄的助手就是变小了的他本人一样，下标操作符应返回它所操作对象的一小部分。对于 text 对象来说，程序员可能希望能够以给定下标(就象它是 C 风格的字符串)来获取单个字符：

```
char & text::operator[](int i)
```

```
{
static char X='';
if((i>0)&&(i<Length))
    return Storage[i];
else
    return X;
}
```

当然，我们希望 text 的下标操作当遇到无效索引时应是安全的。所以，text 返回一个既定空字符来处理这种情况，而无效索引作用于标准 C 字符串时往往会造成系统崩溃。另外要注意的是，函数返回值不是 **char**，而是对 **char** 的引用。这就允许下标元素可以用在赋值操作符的任何一边。常规 C 风格的字符串也是这样的。

但我们不必到此为止。C++并没有限制我们只能使用数字下标！那么，为什么不实现一个查找函数来作为下标呢？它可以在 Basic 语言中象 instr()那样使用：

```
int Position;
text MyText "Can you find this text";
Position=MyText "find this" //Position=8"
```

下面是这个函数的定义：

```
int text::operator[](text SearchFor)const
  {
  char * Test;
  char * Start;
  text Temp;
if(CaseSensitive||SearchFor.CaseSensitive)
      Start=Test=Storage;
  else
      {
      Temp=UpCase();
      Start=Test=Temp Storage;
      SearchFor=SearchFor. UpCase();
      }

  Test=strchr(Test,SearchFor.Storage[0])
  while(Test)
      if(strncmp(Test,SearchFor.Storage,SearchFor.Length)==0)
          return(int)(Test-Start);
      else
          Test=strchr(++Test,SearchFor.Storage[0]);

  return -1;
}
```

如果两个对象都不要求大小写敏感，则生成一个要寻找文本的拷贝；并且两者都变为大写(这里使用了我们还没讲到的一个方法)。接下来，标准 C 库函数 Strchr()和 strncmp()被用来试图确定要寻找的文本在源文本中的位置。如果找到了，则返回该文本距文本头的偏移量；如果没有找到，则返回-1。

但这还没完呢!还记得在头文件的最前面说明和定义的 range 类吗?我说过我们会在后面用到它。这里就用到了。一般情况下，谁都不会费事声明了 range 对象而又将它弃之不用。它一般是用在暂存对象的下标中，如：

```
text SubText;
SubText=MyBigText[range(13,29)];
```

下面呈现了它的实际代码：

```
text text::operator[](range Range)const
    {
    register Count=Range. Last-Range. First+1;
    if((Count<0)||(Count>(Length-Range. First)))
        Count=(Length-Range. First);
    text Temp;
    Temp. Clear();
    Temp. Storage=new char[Count+1];
    memcpy(Temp. Storage,&Storage[Range. First], Count);
    Temp. Storage[Temp. Length=Count]=0;
    return Temp;
    }
```

5.1.7 text 类的加法及加法赋值操作符函数

一个对字符串来说是普通的然而被 C 标准库定义得很糟糕的操作就是 append(附加)。你已有了一个字符串，你想通过在它的尾部接上另一个字符串使它变得更长。这时就需要用 append 操作。这应该是非常容易和高效的，对吧?但一串 append 操作只能使字符串一长再长，而 C 标准库函数 strcat()每次操作都是从字符串头搜寻到尾。另外加之它笨拙的类型处理使之看起来象是由笨妇编写的。

在 Basic、PL/T 以及其它多数自然化的语言中，串联操作都是由加法操作符来完成的。我们在 text 类中就是这样实现的。

一般地，对于既允许赋值又允许非赋值的操作符来说，赋值是最易实现的。下面是它们的说明：

```
class text
{
        ⇓
        ⇓
        text & operator+=(const text & Append);
        text & operator+=(const char * Append)
```

```
        {return *this+=text(Append);}
    text & operator+=(const char Append)
        {return *this+=text(Append);}
    text & operator+=(stringID Append)
        {return *this+=text(Append);}
    ⇓
    ⇓
```

第一个重载操作符可以将另一个 text 对象附加到当前对象上。下面是它的代码：

```
text & text::operator+=(const text & Append)
{
char * Temp=Storage;
Storage=new char[Length+Append.Length+1];
Strcpy(Storage,Temp);
Strcpy(&Storage[Length], Append. Storage);
Length+=Append. Length;
delete[]Temp;
return *this;
}
```

首先，Storage 原来的地址被保存在 Temp 中；接下来为 Storage 重新分配了一个缓冲区，再将原先的值复制到缓冲区中，后面加上要附加的字符串。注意，这里并没有使用 Strcat()函数。幸好我们已经知道了 Length 的值，即附加操作的位置。这样，我们可以调用 Strcpy()函数很容易就完成了操作。

既然赋值加法操作符都能实现了，实现加法操作符就更容易了：

```
class text
{
    ⇓
    ⇓

    text operator+(text Append)const;
    text operator+(const char * Append)const
        {return *this+text(Append);}
    text operator+(const char Append)const
        {return *this+text(Append);}
    text operator+(stringID Append)const
        {return *this+text(Append);}
    friedn text operator+(const char * Original,const text & Append);
    friend text operator+(const char Original,const text & Append);
    friend text operator+(stringID Original,const text & Append);
    ⇓
    ⇓
```

与其它操作符不同，这些操作符使用另一个 text 对象做为参数。但注意，参数并不是 text 对象的引用，而是它的拷贝：

```
text text::operator+(text Append)const
{
text Original(*this);
return Original+=Append;
}

text operator+(const char * Original,const text & Append)
{
return text(Original)+=Append;
}

text operator+(const char Original,const text & Append)
{
return text(Original)+=Append;
}

text operator+(stringID Original,const text & Append)
{
return text(Original)+=Append;
}
```

由于某些原因，Borland 编译器会把这些函数编译成内嵌函数。在这个问题上，我也不想与编译器抗争。一些争斗还是不打的好。

5.1.8 text 类的减法及减法赋值操作符函数

下一个最普通的操作是从 text 字符串中砍掉一块。这可以通过删除特定的子串(如，从"inetallo is very very dangerous"中删除指多余的"very")或取出只基于位置的一片(将机器可识别的文本中的空白转换成数据库操作的入口)来完成。我们已经有许多工具可以迅速完成这种工作。自然地，我们用减法操作符来表示这种操作。下面是它们的说明：

```
class text
{
        ⇓
        ⇓
        text & operator-=(text SearchFor);
        text & operator-=(range Range);

        text operator-(text SearchFor)const;
        text operator-(range Range)const;
        ⇓
```

⇓

而下面就是它们的实现代码：

```
text & text::operator-=(text SearchFor)
{
int Start=(*this)[SearchFor];
if(Start>-1)
      *this=Left(Start)+Mid(Start+SearchFor. Length);
return *this;
}

text & text::operator-=(range Range)
{
*this=Left(Range. First)+Mid(Range. Last+1,Range. Last-Range. First);
  return *this;
}

text text::operator-(text SearchFor)const
{
return text(*this)-=SearchFor;
}

text text::operator-=(range Range)const
{
return text(*this)-=Range;
}
```

这里你还没有见过的函数是 Left()和 Mid()。但如果你曾经用 Basic 编过程序，那么你会认识它们，知道它们是做什么的。(一会儿我们要用 C++实现它们)

5.1.9 text 类的乘法操作符函数

我们常常想要拿一个短的 text 对象，附加上一个、二个或更多的它自己的拷贝。这显然是乘法操作符所要做的事：

```
class text
{
      ⇓
      ⇓
      text & operator*=(int RepeatCount);
      text operator*(int RepeatCount)const;
      ⇓
      ⇓
```

下面是它们的实现：

```
text & text::operator*=(int RepeatCount)
```

```
{
char * Temp=Storage;
int TempLength(Length);
Storage=new char[(Length*=RepeatCount)+1];
register i(0);
for(register c=0;c<RepeatCount;c++);
    {
    memcpy(&Storage[i], Temp, TempLength);
    i+=TempLength;
    }
Storage[i]=0;
delete[]Temp;
return *this;
}

text text::operator*(int RepeatCount)const
{
return text(*this)*=RepeatCount;
}
```

5.1.10 text 类的类型强制转换

类型强制转换(cast)也是一种操作符。我们应该支持类型强制转换，以便构造对象的任何数据类型都可以象下面那样实现：

```
class text
{
    ⇓
    ⇓
    operator const char*() const{return Storage;}
    operator long () const{return atol(Storage);}
    operator double () const{return atof(Storage);}
    ⇓
    ⇓
```

向 long 和 double 的类型转换完成了由构造函数提供的二进制数与文本之间的转换。它们被实现成内嵌函数，所以它们没有额外的函数头。程序员也没有记住象 atol()atof()这样蹩脚函数名的负担了。

通常实现类型强制转换的另一个目的是提供对对象内部隐藏部分的安全获取。但是，向 **const char*** 的类型转换返回的是 Storage 的地址，使用它时要小心一点。因为使用这个类的程序员千万不要用 text 对象代替一个要被写入的 C 风格的字符串。这是因为 text 负责为文本的分配内部缓冲区。但下面这样做是安全的：为缓冲区提供一个指针，通过指针读取数据。

MFC 提供了向 const char*的类型转换；而 Borland 没有提供，但却有一个函数返回 **const**

char* 类型。这里我们仿照 MFC 会更方便一些。

但我们仍需要一种以一些现存的 C 字符串函数来使用 text 对象的方式。我们必须预分配缓冲区并能修改它的大小。这是通过方法 lockBuffer()和 UnlockBuffer()来完成的。所以说，它们可以用在对 Windows API 函数 GetWindowText()的调用中：

```
text Caption;
GetWindowText(Window,Caption.LockBuffer(128), 128);
Caption. UnlockBuffer();
```

下面是这些方法的说明：

```
class text
{
        ⇓
        ⇓
        char* LockBuffer(int RequiredSize);
        void UnlockBuffer(int NewSize=-1);
        long GetLength(void)const{return Length;}
        ⇓
        ⇓
```

LockBuffer()只是使用方法 Store()来分配所需数量的字符：

```
char*text::LockBuffer(int RequiredSize)
{
Store('', RequiredSize);
return Storage;
}
```

对于 UnlockBuffer()，我确实想设计出一个截短缓冲区而不必复制字符串的方法，但我没做到。我同样也考虑了让那些额外的、不用的字节就那样挂着，但看起来那样也不对。所以，我们最终还是重新分配文本缓冲区：

```
void text::UnlockBuffer(int NewSize)
{
if(NewSize==-1)
        NewSize=strlen(Storage);
if(NewSize<Length)
        {
        char * Temp=Storage;
        Storage=new char[NewSize i 1];
        strcpy(Storage,Temp);
        Length=Newsize;
        delete[]Temp;
        }
}
```

5.1.11 text 类的切分和大小写变换

我们已经见过了 Left()和 Mid()这两个方法(在减法操作符中使用过)。既然存在着一个 Left(),你可能想到还会有一个 Right()。

```
text text::Left(int Count)const
{
if(Count>Length)Count=Length;
text Temp;
Temp,Clear();
Temp. Storage=new char[Count + 1];
memcpy(Temp,Storage,Storage,Count);
Temp. Storage[Temp. Length=Count]=0;
return Temp;
}

text text::Right(int Count)const
{
if(Count>Length)Count=Length;
text Temp;
Temp. Clear();
Temp. Storage=new char[Count+1];
memepy(Temp. Storage,&Storage[Length-Count], Count);
Temp. Storage[Temp. Lengin=Count]=0;
return Temp;
}
```

就象在 Basic 中那样,Left()返回位于左端的 Count 个字符的子串;Right()返回位于右端的 Count 个字符的子串。

大家都知道我是怎样地讨厌缩写,但 Mid()作为"middle"的缩写已经使用很长时间了。所以在这里我沿用了它。它的实现使用了下标操作符:

```
text text::Mid(int Start,int Count)const
{
return(*this)[range(Start,Start+Count-1)];
}
```

你也可以将文本分成大写的和小写的。当人们更喜欢大写或小写文本时,他们往往很在意文本的大小写。C 标准库包含有转换大小写字符的函数(它们实际上是宏)。我们可以使用它们:

```
text text::UpCase(void)const
{
text Temp(*this);
for(register i=0;i<Temp. Length;i++)
```

```
    Temp. Storage[i]=toupper(Temp.Storage[i]);
return Temp;
}
```

```
text text::DownCase(void)const
{
text Temp(*this);
for(register i=0;i<Temp.Length;i++)
    Temp. Storage[i]=tolower(Temp.Storage[i]);
return Temp;
}
```

我们可以就此止步了，但现在对一种叫做"title case"的大小写格式的需要越来越多。这里，函数名中每个词的首字符设置成大写而其它字符设置成小写。如果不是在用面向对象的思想进行程序设计，程序员就不必考虑编写这种操作的代码。下面我们定义了TitleCase()方法：

```
text text::TitleCase(void)const
{
text Temp(*this);
BOOL AfterSpecial(TRUE);
Temp. DownCase();
for(register i=0;i<Temp.Length;i++)
    if(! isalnum(Temp.Storage[i]))
        AfterSpecial=TRUE;
    else if (AfterSpecial);
        {
        Temp.Storage[i]=toupper(Temp.Storage[i]);
        AfterSpecial=FALSE;
        }
return Temp;
}
```

这部分的代码在文本缓冲区中进行循环，当遇到非字母字符或非数字字符时设置相应的标志。幸好UpCase()、DownCase()和TitleCase()并不需要重新分配文本缓冲区，转换是就地进行的。

5.2 测试你的新实力

面向对象程序设计很棒的一方面就是你可以真正地验证你的应用程序。如果程序的各部分都正确地工作。那么整个程序也就会正确地工作。所以应该花一些时间来测试你编写的方法。这种做法当然要比跟踪错误调试程序更加有用。

一个恰当的测试程序一般要单独运行每一个函数，确保输入的数据通常要获得期望的数据输出。但如果类的绝大多数方法只是执行转换(就象在text类中那样)，你则可以对你的类

充满信心，只需编写一个大的工作函数来测试你的类就可以了。为了使测试程序的代码量最小，我在这里提供了下面的程序。这个程序在 Borland 或 Microsoft 编译器中应分别使用“Easy-Win”或“QuickWin”选项来进行编译：

```
#include<windows.h>
#include "texth"
#include<iostream.h>

void main(void)
    {
    text Test1("This is a test string.");
    cout<<(const char*)Test1<<endl;

    Testl="New text by assignment";
    cout <<(const char*)Text1<<endl;

    test1+= "..."
    cout<<(const char*)Test1<<endl;

    Test1=Test1.UpCase();
    cout<<(const char*)Test1<<endl;
    Test1=Test1.DownCase();
    cout<<(const char*)Test1<<endl;
    Test1=Test1.TitleCase();
    cout<<(const char*)Test1<<endl;

    Test1="Quoting a string";
#ifdef MSC VER
    Test1=(const char *) "\" +Test1+(char) "\";
#else
    Test1= "\" +Test1+(char) "\";
#endif
    cout<<(const char *)Test1<<endl;

    Test1=781L;
    Test1+="L";
    cout<<"As string:"<<(const char *)Test1<<endl;
    cout<<"As double:"<<(double)Test1<<endl;

    Test1="Your wretched refuse yearning to breathe free…";
    cout<<(const char *)Test1<<endl;
```

```
#ifdef _MSC_VER
    cout<<"'refuse' at position"<<Test1[text("refuse")] <<endl;
    cout<<"'REFUSE'at position"<<Test1[text("REFUSE")]<<endl;
    Test1.SetCaseSensitive();
    cout<<"'REFUSE'at position"<<Test1[text("REFUSE")] <<endl;
#else
    cout<<"'refuse'at position"<<Test1["refuse"]<<endl;
    cout<<"'REFUSE'at position"<<Test1["REFUSE"]<<endl;
    Test1.SetCaseSensitive();
    cout<<"'REFUSE'at position"<<Test1["REFUSE"]<<endl;
#endif

    cout<<"'"<<(const char *)test1.Left(4)<<"'"<<endl;
    cout<<"'"<<(const char *)Test1.Right(7)<<"'"<<endl;
    cout<<"'"<<(const char *)Test1.Mid(5,8)<<"'"<<endl;

    Test1-="refuse";
    cout<<(const char *)Test1<<endl;

    Test1_=range(33,37);
    cout<<(const char *)Test1<<endl;
    cout<<(const char *)Test1[range(5,12)] <<endl;

    Test1="Ha";
    Test1*=10;
    cout<<(const char *)Test1<<endl;
    }
```

你会注意到有两个地方使用了#**define**语句来区分Visual C++和Borland C++，两种编译器所支持的语法格式略有不同。Visual C++1.52不能完全兼容地处理ANSI标准重载操作符，它需要额外的帮助来确定到底使用的是哪个操作符。

为了节省代码空间，我略掉了一些简单函数的测试。那些函数我确信它们是无错的。

第六章　实现VBX++类库

在第四章和第五章我们创建了两个类库：与软件商无关的GDI库和text库。现在我们有理由来编写一个C++的VBX了，并且我们已经编写了相应的工具。那么，我们要建造的到底是什么东西呢?使用图表或许会有助于了解释它。图6.1说明了我们系统的体系结构。

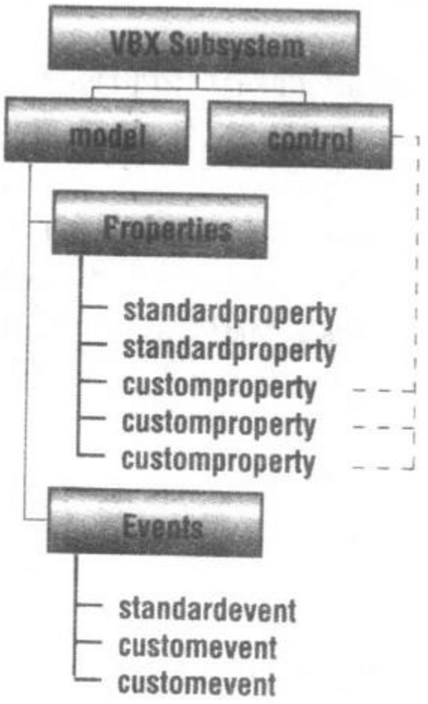

图6.1　VBX子系统的体系结构

VBX++是用来创建VBX的C++类库。它要被设计成与软件商无关的，即在Visual C++或Borland C++下都可以编译。它包含有下面几个主要的类：

●model类：它是一个基类。你可以基于这个类为你的VBX所支持的用户控件派生出一个新类。(VBX可以包含多个控件。)model类是属性和事件的容器。它所包含的属性和事件是在它的构造函数中加入的。你可以为你的派生类创建一个实例。

●property和event类：这两个类的对象描述控件的属性和事件，它们要被加到model容器中。这里我们也提供了预定义的标准属性和事件对象。

●control类：这个类的派生对象代表控件的运行拷贝。在这里完成消息的处理。你不要自己去创建这个类的实例。它们是当底层控件创建时被动态生成的。用户属性要依赖control类处理数据。

6.1　使用VBX++：一次预览

在我们弄清楚这些类是如何实现以前，让我们先来大致看看它们是如何使用的。假设你想要做一个叫做“Gradient”的用户控件。这个图形控件要支持两个属性，但不支持事件。这样，代表这个控件的类就可以从model类派生出来，并可以命名为gradient：

```
class gradient: public model
```

```
{
public:
    gradient();
protected:
    virtual FLONG GetModelFlags(void);
    vittual void OnDraw(control& Control,devicecontext& Dc,
        BOOL Erase, rectangle Rect);
private:
    colorproperty TopColorProp;
};
```

类的构造函数的一个任务就是注册(即加入)控件的属性和处理对象的方法。它看起来就是一个简单的容器:

```
gradient::gradient()
  :     model((PSTR) "Gradient"), TopColorProp((PSTR) "TopColor", RGB(255,
        255,255))
  {
  Properties+ = *Top;
  Properties+ = *Left;
  Properties+ = *Height;
  Properties+ = *Width;
  Properties+ =TopColorProp;
  }
```

你要注意标准属件和用户属性的使用。标准属性是用指针来表示的。如果有事件，事件也要在这里加入(到 Events 数据成员中)。还要注意用户属性不是从一般的 property 派生出来的，而是从派生类 colorproperty 派生出来的。并且 colorproperty 类指定了属性的数据类型。

方法 GetModelFlags()是虚函数(**virtual**)。它重载了基类中的同名方法，而且它唯一的任务就是返回一个 MODEL 标志:

```
FLONG gradient::GetModelFlags(void)
  {
  return MODEL_fGraphical;
  }
```

方法 OnDraw()使用了我们在第四章开发的小型 GDI 类库中定义的设备环境和矩形类:

```
void gradient::OnDraw(control & Control,
        devicecontext & Dc,BOOL Erase,rectangle Rect)
  {
  color TopColor((COLORREF)TopColorProp. GetValue(Control))
  color LineColor,
  register Percent;

  register Offset(Rect. top);
```

```
Rect. top=0;
Rect. bottom-=Offset;

for(register i=0;i<Rect. bottom;i++)
    {
    Percent=100-((i*100)/Rect. bottom);
    LineColor=
        red((BYTE)((red)TopColor)*(BYTE)Percent/100);
    LineColor=
        green((BYTE)((green)TopColor)*(BYTE)Percent/100);
    LineColor=
        blue((BYTE)((blue)TopColor)*(BYTE)Percent/100);
    pen Pen(LineColor,2,pen::InsideFrame);
    Dc=Pen;
    Dc. MoveTo(point(Rect. left,Offset+i));
    Dc. LineTo(point(Rect. right,Offset+i));
    }
}
```

一旦定义了 model 的派生类，你就可以生成一个这个类的实例：

```
gradient Gradient;
```

嘿，这就做完了!编译它你就可以获得一个 VBX 用户控件。不惊讶吗?

6.2 VBX++的实现

Microsoft Visual C++和 Borland C++都支持预编译头。预编译头会节省大量的编译时间。为了形成预编译头，VBX++的各个模块都单独包含一个 VBXXX. H 文件，而这个头文件再包含其它头文件：

```
#ifndef_VBXXX_H_
#define_VBXXX_H_

#define STRICT
#include<windows. h>

#ifndefVB_VERSION
#include<vbapi. h>
#endif

#include"text. h"
#include"portagdi. h"
#include"control. h"
```

```
#include"events.h"
#include"props.h"
#include"model.h"

#endif
```

为了使这个头文件只能被一个模块包含一次，我使用了编译宏指令：

```
#ifdef_VBXXX_H_
#define_VBXXX_H_
```

以及结尾的#endif 语句。很糟糕的是，Microsoft 公司在 VBAPI.H 文件中并没有这么做。如果你不小心重复包含这个头文件的话，你就会惹出一大堆“重复定义”的错误。

TEXT H 和 PORTAGDI.H 这两个文件是 VBX++所要用到的两个单独类的头文件。并且这两个类的所有的模块都没有包含 VBXXX.H 文件，因为它们是单独的类。它们只分别需要 TEXT.H 和 PORTAGDI.H 头文件。其它的头文件都是 VBX++的组成部分。

VBX++被组装成带头文件的类库。这就是说，我们的工程目标是 LIB 文件。

一般来说，当我们开发多类工程时，要把类划分成几个模块分别放到不同的文件中去定义。但是如果各模块联系得比较紧密，特别是当这些模块必须要一起使用时，还是将它们放在一个文件中为好。当然，这并意味着 PORTAGDI 部分就非要挤到 VBXXX.CPP 文件中(VBXXX.CPP 文件中定义了一组相关的类)。但我们下面要学习的那些类都是在 VBXXX.H 文件中进行说明，并且在 VBXXX.CPP 文件中进行定义的。

6.2.1 Microsoft Visual C++编译器选项设置

在创建这个工程时，你必须记住两件事情：第一件是你的目标类型是“静态库”；第二件是不要设置“Use Microsoft Foundation Classes”的选项。创建工程的对话框应该或多或少地类似于图6.2所显示的那样。

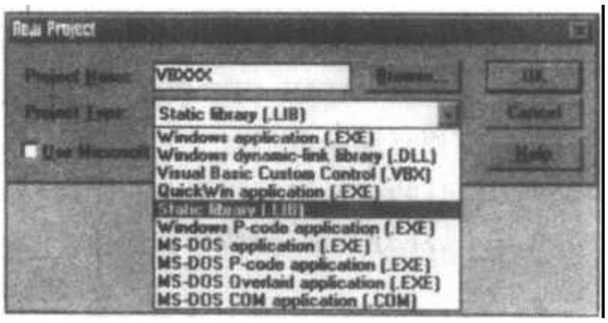

图6.2 Vistual C++中生成 VBX++工程的 New Project 对话框的设置选择

在创建了新工程后，要把各模块加入到工程中去。此外，你还需要改变一些编译器的设置，如：

1. 尽管这是一个 LIB 工程，你也要使用大模式(Large Model)来编译，因为这个库最终要用于建造 VBX，它是一种 DLL。

2. 可以设置 Code Generator 中的 CPU 选项为80386 CPU 或更高级的 CPU。Windows 3.1 现在或多或少有点过时了，它是支持80286 CPU 的最后一个 Windows 版本。

3. 在预编译头(Precomplied Header)选项中，设置“Automatic Use of Precomplied Headers”，但不要输入 VBXXX.H 文件名。如果你输入了这个文件名，MSVC 在编译

TEXT.CPP 和 PORTAGDI.CPP 模块时就会出错，因为这两个模块中并没有包含这个头文件。

4. 在 Windows Prolog/Epilog 选项中，选择"Protected Mode DLL Functions"。

6.2.2 Borland C++编译器选项设置

在创建一个新工程时，你必须记住两件事情：第一件是你的目标类型是"静态库"；第二件是不要选择"OWL"或"Class Library"设置。创建新目标对话框应或多或少类似于图6.3所显示的那样。

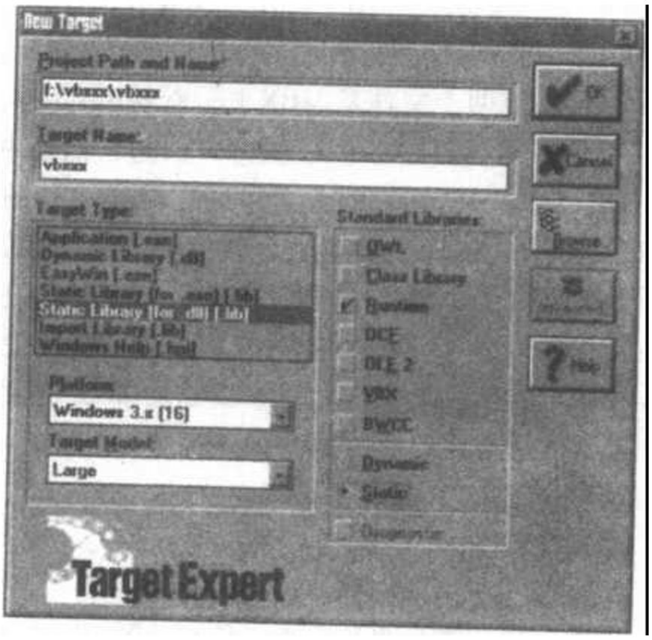

图6.3 Borland C++中生成 VBX++工程的 New Target 对话框的设置选择

在创建了新工程后，要把各模块加入到工程中去。此外，你还需要改变一些编译器的设置，如：

1. 在 Precomplied Header 选项中，在"Stop Precompiling after header file"下输入 VBXXX.H 文件名。TEXT.CPP 和 PORTAGDI.CPP 模块中没有包含这个头文件，就不使用预编译头，而其它的模块则可以使用预编译头。

2. 可以设置16-Bit Compiler 中的 Processor 选项为80386CPU。

3. 在16-Bit Compiler 中的 Entry/Exit Code 选项中，选择"Windows DLL，Explicit Functions Exported"。

6.2.3 创建 model 类

VBX++是通过将 CDK 结构封装成类来与 Visual Basic CDK 的"C"世界挂上钩的，比如：MODEL 结构是 model 类的一个属性；PROPINFO 结构是由 standardproperty 类来管理的等等。这里，我们来学习 model 类。

model 类的说明是相当长的，我把它全部呈现在这儿。这样，你就可以看到它的全部。但对于类的每个成员，我们在实现它们时再讨论：

```
#ifndef _MODEL_H_
#define _MODEL_H_
```

```
ciass model
    {
    //Management of static list of instantiated models
    //   including registration
    friend BOOL far pascal_export VBINITCC(USHOT,BOOL);
    friend VOID far pascal_export VBTERMCC(void);
    public:
        enum {MaxModels=12};
    private:
        static UINT ModelCount;
        static model* ModelList[MaxModels];
        slatic USHORT Run Version;
        static BOOL Runtime;
        MODEL Model;
        static BOOL RegisterModels(USHORT aRunVersion,BOOL aRuntime);
        BOOL RegisterModel();
        void SetModelData(void);
    protected:
        virtual USHORT GetModelVersion(void);
        virtual FLONG GetModelFlags(void);
        virtual FSHORT GetClassStyles(void);
        virtual FLONG GetWindowStyles(void);
        virtual WORD GetButtonBitmapBaseID(void);
        virtual PSTR GetDefaultControlName(void);
        virtual PSTR GetParentClassName(void);
        virtual BYTE GetDefaultPropertyIndex(void);
        virtual BYTE GetDefaultEventIndex(void);
        virtual BYTE GetValuePropertyIndex(void);
        virtual BYTE GetControlVersion(void);

    //Construction of model, including standard properties and events public:
        model(LPSTR aName);
        propertieslist Properties;
        eventslist Events;
    private:
        char ClassName [64];
        static standardproperty* Name;
        static standardproperty* Index;
    protected:
        static standardproperty* Hwnd;
```

```
        static standardproperty* BackColor;
        static standardproperty* ForeColor;
        static standardproperty* Left;
        static standardproperty* Top;
        static standardproperty* Width;
        static standardproperty* Height;
        static standardproperty* Enabled;
        static standardproperty* Visible;
        static standardproperty* MousePointer;
        static standardproperty* Caption;
        static standardproperty* FontName;
        static standardproperty* FontBold;
        static standardproperty* FontItalic;
        static standardproperty* FontStrike;
        static standardproperty* FontUnder;
        static standardproperty* FontSize;
        static standardproperty* TabIndex;
        static standardproperty* Parent;
        static standardproperty* DragMode;
        static standardproperty* DragIcon;
        static standardproperty* BorderStyleOff;
        static standardproperty* TabStop;
        static standardproperty* Tag;
        static standardproperty* Text;
        static standardproperty* BorderStyleOn;
        static standardproperty* ClipControls;
        static standardproperty* None;
        static standardproperty* HelpContextID;
        static standardproperty* LinkMode;
        static standardproperty* LinkItem;
        static standardproperty* LinkTopic;
        static standardproperty* LinkTimeout;
        static standardproperty* LeftNoRun;
        static standardproperty* TopNoRun;
        static standardproperty* Align;
        static standardproperty* ImeMode;
        static standardproperty* DataSource;
        static standardproperty* DataField;
        static standardproperty* DataChanged;
    private:
```

```
        static void ConstructStandardProperties(void);
protected:
        static standardevent * Click;
        static standardevent * DblClick;
        static standardevent * DragDrop;
        static standardevent * DragOver;
        static standardevent * GotFocus;
        static standardevent * KeyDown;
        static standardevent * KeyPress;
        static standardevent * KeyUp;
        static standardevent * LostFocus;
        static standardevent * MouseDown;
        static standardevent * MouseMove;
        static standardevent * MouseUp;
        static standardevent * LinkError;
        static standardevent * LinkOpen;
        static standardevent * LinkClose;
        static standardevent * LinkNotify;
        static standardevent * LinkChange;
        static standardevent * ObsoleteEvent;
private:
        void ConstructStandardEvents(void);
//Destructor
public:
        ~model();
private:
        void DestroyStandardProperties(void);
        void DestroyStandardEvents(void);
//Message dispatching
      friend  LRESULT  CALLBACK_ export  CuProc ( HCTL, HWND, USHORT,
      USHORT,long);
        controlbag Controls;
        static model * FindModel(HCTL Handle);
        LRESULT Dispatch(control&.,USHORT,USHORT,long);
protected:
        virtual LRESULT OnInitialize(control &. Control);
        virtual void OnActivate(control &. Control,UINT active,)
                BOOL minimized, HWND hWndOther);
        virtual void OnCancelMode(control &. Control);
        virtual void OnChar(control &. Control,UINT Key,
```

```
        UINT RepeatCount,UINT flags);
    virtual int OnChatToItem(control &. Control,UINT Key,
        HWND hWndListBox,UINT caretPos);
    virtual void OnClose(control &. Control);
    virtual LRESULT OnCompareItem(control &. Control,
        UINT ControlID,COMPAREITEMSTRUCT far&. compareInfo);
virtual int OnCreate(control &. Control,
    CREATESTRUCT far &. Create);
virtual HBRUSH OnCtlColor(control &. Control,devicecontext &. Dc,
    HWND Child,UINT ControlType);
virtual void OnDeadChar(control &. Control,UINT deadKey,
    UINT RepeatCount,UINT flags);
virtual void OnDeleteItem(control &. Control,UINT ControlID,
    DELETEITEMSTRUCT far&. deleteInfo);
virtual void OnDestroy(control &. Control);
virtual void OnDrawItem(control &. Control,UINT ControlID,
    DRAWITEMSTRUCT far&. drawInfo);
virtual void OnDropFiles(control &. Control,HANDLE DropInfo);
virtual void OnEnable(control &. Control,BOOL Enabled);
virtual BOOL OnEraseBkgnd(controi &. Control,devicecontext &. Dc);
virtual HFONT OnGetFont(control &. Control);
virtual void OnGetMinMaxInfo(control &. Control,
    MINMAXINFO far &. Info);
virtual void OnGetText(control &. Control,UINT MaxLength,
    LPSTR Buffer);
virtual UINT OnGetTextLength(control &. Control);
virtual void OnHScroll(control &. Control,UINT ScrollCode,
    UINT ThumbPos,HWND hWndCtl);
virtual void OnKeyDown(control &. Control,UINT Key,
    UINT RepeatCount,UINT flags);
virtual void OnKeyUp(control &. Control,UINT Key,
    UINT RepeaiCount,UINT flags);
virtual void OnKillFocus(control &. Control,HWND hWndGetFocus);
virtual void OnLButtonDblClk(control &. Control,UINT Flags,
    point Point);
virtual void OnLButtonDown(control &. Control,UINT Flags,
    point Point);
virtual void OnLButtonUp(control &. Control,UINT Flags,
    point Point);
virtual void OnMButtonDblClk(control &. Control,UINT Flags,
```

```
        point Point);
    virtual void OnMButtonDown(control & Control,UINT Flags,
        point Point);
    virtual void OnMButtonUP(control & Control,UINT Flags,
        point Point);
    virtual void OnMeasureItem(control & Control,UINT ControlID,
        MEASUREITEMSTRUCT far & measureInfo);
    virtual void OnMouseMove(control & Control,UINT Flags,
        point Point);
    virtual void OnMove(control & Control,point clientOrigin);
    virtual UINT OnNCHitTest(control & Control,point Point);
    void OnPaint(control & Control);
    virtual void OnDraw(control & Control,devicecontext & Dc,
        BOOL Erase,rectangle Rect);
    virtual int OnPower(control & Control,UINT PowerEvent);
    virtual BOOL OnQueryEndSession(control & Control);
    virtual void OnRButtonDblClk(control & Control,UINT Flags,
        point Point);
virtual void OnRButtonDown(control & Control,UINT Flags,
    point Point);
virtual void OnRButtonUp(control & Control,UINT Flags,
    point Point);
virtual BOOL OnSetCursor(control & Control,HWND hWndCursor,
    UINT hitTesl,UINT mouseMsg);
virtual void OnSetFocus(control & Control,HWND hWndLostFocus);
virtual void OnSetFont(control & Control,HFONT hFont,
    BOOL redraw);
virtual void OnSetText(control & Control,LPSTR Buffer);
virtual void OnSize(control & Control,size Size);
virtual void OnSpoolerStatus(control & Control,
    UINT JobStatus,UINT JobsLeft);
virtual void OnSysColorChange(control & Control);
virtual void OnTirneChange(control & Control);
virtual void OnTimer(control & Control,UINT TimerID);
virtual vint OnVKeyToItem(control & Control,UINT Key,
    HWND ListBox,UINT caretPos);
virtual void OnVScroll(control & Control,UINT ScrollCode,
    UINT ThumbPos,HWND hWndCtl);
virtual void OnWinIniChange(control & Control,LPSTR Section);
virtual void VbCancelMode(control & Control,HWND CapturingWindow);
```

```
    void VbPaint(control & Control,HDC dc,LPRECT pRect);
    LRESULT OnMethod(control & Control,WORD MethodID,LONG lParam);
    virtual LRESULT OnMethodAddItem(control & Control,
        LPSTR Item,int Index);
    virtual LRESULT OnMethodClear(control & Control);
    virtual LRESULT OnMethodDrag(control & Control,int Command);
    virtual LRESULT OnMethodLinkSend(control & Control);
    virtual LRESULT OnMethodMove(control & Control,
        int Count,LONG Left,LONG Top,LONG Width,LONG Height);
    virtual LRESULT OnMethodRemoveItem(control & Control,int Index);
    virtual LRESULT OnMethodRefiesh(control & Control);
    enum ZOrderCommand {BringToFront, SendToBack};
    virtual LRESULT OnMethodZOrder(control & Control,
        ZOrderCommand Command);
  };
#endif
```

当然，占去大量篇幅的是那些各种消息处理函数的原型。现在，让我们来看看 MODEL. CPP 文件的内容：

有意思的是，在被大量的类说明吓了一跳之后，我们在 MODEL. CPP 文件中发现的第一个函数并不是 model 类的成员函数！为了使程序员能毫不费力地使用 VBX++，我们必须提供一些“C”类函数。例如，每个 DLL(注意：VBX 就是一种 DLL)必须有一个 LibMain()函数。你知道 C++中可以对函数用同一个函数名进行重新定义，即重载。但 LibMain()函数不能被重载。避免重载的唯一办法是在 extern"C"语句中说明函数原型。好在 LibMain()函数在 WINDOWS. H 文件中就是以这种方式预说明的。我们这里的 LibMain()的主要操作是获取和保存 VBX 的实例句柄：

```
#inciude "model. h"
HANDLE LibInstance=0;
extern"C"int far pascal LibMain(HINSTANCE Instance,WORD Dalaseg,WORD HeapSize,
LPSTR Commandline)
  {
  if(HeapSize>0)UnlockData(0);
  LibInstance=Instance;
  return TRUE;
  }
```

我们还将需要两个 CDK 函数：VBINITCC()和 VBTERMCC()。同样这两个函数也要说明或定义成不能重载的方式。我们把它们说明为 model 类的友元函数。下面是它们的定义：

```
extern"C"BOOL far pascal_export VBINITCC(USHORT Version,BOOL Runtime)
  {
  return model::RegisterModels(Version,Runtime);
  }
```

```
extern"C"VOID far pascal_export VBTERMCC(void)
{
}
```

通常 VBINITCC()的任务是注册每个用户控件都包含的 VBX 中的 MODEL 结构。就是在这个函数中我们通过调用 model 类的静态函数 RegisterModels()成员来使 VBX 与 model 类建立连接的。

实际上你是通过静态生成一个 model 类的实例来使用 model 对象的。这意味着 model 类必须维护一个 model 对象指针的列表。这个列表作为一个静态属性被所有 model 对象所共享：

```
UINT model::ModelCount=0;
model* model::ModelList [model::MaxModels];
USHORT model::RunVersion;
BOOL model: Runtime;
```

过一会儿我们就会看到，对象在生成时就将自己加入到这个列表中。RegisterModels()方法必须遍历这个列表，告诉表中的每个 model 去注册自身：

```
BOOL model::RegisterModels(USHORT aRunVersion,BOOL aRuntime)
  {
  RunVersion=aRunVersion;
  Runtime=aRuntime;
  for(register UINT i=0;i<ModelCount;i++)
      if(! ModelList[i]->RegisterModel())
          return FALSE;
      return TRUE,
  }
```

每个 model 对象通过填写它的 MODEL 结构和调用 VBRegisterModel()函数来注册它自己：

```
BOOL model::RegisterModel()
  }
  SetModelData();
  return VBRegisterMODel(LibInstance,&Model);
  }
```

SetModelData()函数的功能是确保 MODEL 结构在被用于注册前结构中的数据的值是否合适。它是通过调用一组虚函数来完成的；并且对每个结构域至少调用了一个虚函数：

```
struct VBDATA
  {
  model far*Model;
  }
typedefVBDATA far*LPVBDATA;

void model::SetModelData()
  {
```

```
    Model. usVersion=GetModelVersion();
    Model. fl=GetModelFlags()|MODEL_fInitMsg;
    Model. pctiproc=Ctlproc;
    Model. fsClassStyle=GetClassStyles();
    Model. flWndStyle=GetWindowStyles();
    Model. cbCtlExtra=sizeof(VBDATA);
    for(register unsigned i=0;i<Properties. GetCount(); i++)
        {
        register UINT Size(Properties[i]. GetStorageSize());
        if(Size)
            Properties[i]. SetStorageOffset(Model. cbCtlExtra);
        Model. cbCtlExtra+=Size;
        }
  Model. idBmpPalette-GetButtonBiunapBaseID();
  Model. npszDefCtlName=GetDefaultControlName();
  Model. npszParentClassName=GetParentClassName();
  Model. npproplist=Properties. GetList();
  Model. npeventlist=Events. GetList();
  Model. nDefProp=GetDefaultPropertyIndex();
  Model. nDefEvent=GetDefaultEventIndex();
  Model. nValueProp=GetValuePropertyIndex();
  switch(Model. usVersion)
      {
      case 100: Model. usCtlVersion=1; break;
      case 200: Model. usCtlVersion=2; break;
      default: Model. usCtlVersion=GetControlVersion();
      }
    }
```

尽管大部分域是直接赋值的，但有一些仍需进一步解释。比如，要填写 Model. fl 域的值，函数调用了 GetModelFlags()，但它还需要 MODEL_fInitMsg 标志的值。(这个标志使控件生成时，会接收到一个 VBM_INITIALIZE 消息。后面你就会看到，当控件生成时，我们必须要接收处理这样一些消息。)CtlProc()是类的友元函数，它是控件的控制过程(即窗口过程)。

Model. cbCtlExtra 域必须要包含一定的空间来存贮控件的用户属性。**for** 循环语句是通过获取 model 类的 Properties 数组中的每个属性大小来计算 Model. cbCtlExtra 域所需空间大小的。最后，如果程序员需要指定其它版本号(而不是默认的300)，Model. usCtlVersion 域可以调整为其它值。对于 VB1. 0和2. 0来说，这个域的用处是不同的。

那些虚函数都是直接返回的，它们返回域的默认值。如果需要的话，程序员可以重载这些函数以提供其它的值：

```
USHORT model::GetModelVersion(void)
  {
```

```
    return VB300_VERSION;
    }
FLONG model::GetModelFlags(void)
    {
    return MODEL_fFocusOk|MODEL_fArrows;
    }
FSHORT model::GetClassStyles(void)
    {
    return CS_VREDRAW|CS_HREDRAW;
    }
FLONG model::GetWindowStyles(void)
    {
    return WS_BORDER;
    }
WORD model::GetButtonBitmapBaseID(void)
     {
     return 8000;
     }
PSTR model::GetDefaultControlName(void)
    {
    return Model.npszClassName;
    }
PSTR model::GetParentClassName(void)
    {
    return NULL;
    }
BYTE model: GetDefaultPropettyIndex(void)
    {
    return 0;
    }
BYTE model::GetDefaultEventIndex(void)
    {
    return 0;
    }
BYTE model::GetValuePropertyIndex(void)
    {
    return-1;
    }
BYTE model::GetControlVersion(void)
    {
```

```
    return 100;
    }
```

以上那些函数直接或间接地支持 model 类的静态属性。现在我们可以看看 model 类的构造函数：

```
model::model(LPSTR aClassName)
  {
  if(!ModelCount)
      {
      ConstructStandardProperties();
      ConstructStandardEvents();
      }
  if(ModelCount<MaxModels)
      ModelList[ModelCount++]=this;
  Model.npszClassName=aClassName;
  Properties+=*Name;
  Properties+=*Index;
  Properties+=*Parent;
  }
```

正如所希望的，model 对象将自己加入到静态 ModelList 属性中。但在加入之前，它要检查它是否是被注册的第一个 model 对象。如果是的话，它要构造标准的属性和事件。这些标准的属性和事件也是类的静态成员。最后，通常将“Name”和“index”属性被加入到 Properties 容器中。

实现文件中下面的部分定义了标准属性的初始值以及初始化它们的方法：

```
standardproperty*model::Name=NULL;
standardproperty*model::Index=NULL;
standardproperty*model::Hwnd=NULL;
standardproperty*model::BackColor=NULL;
standardproperty*model::ForeColor=NULL;
standardproperty*model::Left=NULL;
standardproperty*model::Top=NULL;
          ⇓
          ⇓
standardproperty*model::DataSource=NULL;
standardproperty*model::DataField=NULL;
standardproperty*model::DataChanged=NULL;
  void model::ConstructStandardProperties(void)
      {
      Name=new standardproperty(PPROPINFO_STD_NAME);
      Index=new standardproperty(PPROPINFO_STD_INDEX);
      Hwnd=new standardproperty(PPROPINFO_STD_HWND);
```

```
BackColor=new standardproperty(PPROPINFO_STD_BACKCOLOR);
ForeColor=new standardproperty(PPROPINFO_STD_FORECOLOR);
Left=new standardproperty(PPROPINFO_STD_LEFT);
Top=new standardproperty(PPROPINFO_STD_TOP);
        ⇓
        ⇓
DataSource=
    new standardproperty(PPROPINFO_STD_DATASOURCE, VB300_VERSION);
DataField=
    new standardproperty(PPROPINFO_STD_DATAFIELD, VB300_VERSION);
DataChanged=
    new standardproperty(PPROPINFO_STD_DATACHANGED,VB300_VERSION);
}
```

这已不是我第一次的选择了。我想简单将对象放在数据段中,让系统在执行主模块之前运行对象的构造函数。但不幸的是,试验表明我不能保证在 model 对象的构造函数之前调用属性的构造函数，而这是需要的。所以我们使用 new 操作符来生成标准事件:

```
standardevent* model::Click=NULL;
standardevent* model::DblClick=NULL;
standardevent* model::DragDrop=NULL;
        ⇓
        ⇓
standardevent* model::ObsoleteEvent=NULL;

void model::ConstructStandardEvents(void)
    {
    Click=new standardevent(PEVENTINFO_STD_CLICK);
    DblClick=new standardevent(PEVENTINFO_STD_DBLCLICK);
    DragDrop=new standardevent(PEVENTINFO_STD_DRAGDROP);
        ⇓
        ⇓
    ObsoleteEvent=
        new standardevent(PEVENTINFO_STD_NONE,VB200_VERSION);
    }
```

model 对象的析构函数必须删除(delete)掉构造函数所生成的所有标准属性，但只有当最后一个 model 派生类撤消时，才可这样做:

```
model::~model()
    {
```

```
    if(!(--ModelCount))
        {
        DestroyStandardProperties();
        DestroyStandardEvents();
        }
    }
```

实际用于从堆中删除那些对象的方法虽然是冗长的，但却是必要的：

```
void model::DestroyStandardProperties(void)
  {
  delete Name;
  delete Index;
  delete Hwnd;
  delete BackColor;
  delete ForeColor;
  delete Left;
  delete Top;
       ⇓
       ⇓
  delete DataSource;
  delete DataField;
  delete DataChanged;
  }

void model::DestroyStandardEvents(void)
   {
  delete Click;
  delete DblClick;
  delete DragDrop;
       ⇓
       ⇓
  delete ObsoleteEvent;
  }
```

既然我们已经知道了 model 对象是如何被加入到对象的静态表中的、是如何注册的以及是如何构造和析构的(伴随着标准属性和事件的构造和析构)，接下来我们可以看看它是如何被使用的了。

你一定还记得，就象所有的窗口都有一个窗口过程一样，所有的 VBX 控件也都有一个控制过程。它们两个非常相似。实际上，那些发送到窗口过程的消息在控件中被发送到控制过程。其中的差别就是控制过程要接受一些额外的消息。大多数这类消息都有 VBM_ 前缀。

有一个问题是，当控制过程接收到 一个消息时，它获得的是控件句柄，而不是控件对象的指针，当然也不是 model 对象的指针。我们必须解决这个问题。下面让我们看看控件过程：

```
LRESULT CALLBACK export CtlProc
        (
        HCTL Handle,
        HWND Window,
        USHORT Msg,
        USHORT wParam,
        long lParam
        )
 {
LPVBDATA VbData=(LPVBDATA)VBDerefControl(Handle);
control*Control=NULL;

if(Msg==VBM_INITIALIZE)
     {
     VbData->Model=model::FindModel(Handle);
     Control=new control(Handle,VbData->Model)
     VbData->Model->Controls+=*Control;
     VbData->Model->OnInitialize(*Control);
     return 0;
     }
else if(VbData)
    Control=VbData->Model->Controls.Find(Handle);
```

⇓

⇓

每个控件实例都要多次调用这个过程，除了第一次外，在其它后续调用中，我们可以简单地通过查寻控件列表来获得相关联的控件对象。但第一次调用是不同的，因为这时还没有与其相关联的控件对象。幸运的是，由于我们在 MODEL 中设置了相关的标志，所以我们会获得特殊的第一个消息：VBM_INITIALIZE 消息。在响应这个消息时，我们获得这个控件所属的 model，并生成代表它的控件对象。

虽然我们希望大部分处理都发生在控件方法内部，但除了 VBM_INITIALIZE 消息外，还有一对消息要特殊处理。其中一个是 WM_NCCREATE 消息；控件实际上也是一种窗口，因此，在控件生成时会收到这个消息。假设这个消息的默认处理成功的话，我们使用这个消息来保存与控件相关联的窗口句柄。如果控件是图形控件，那么这个句柄就为 NULL：

⇓

⇓

```
if(Msg==WM_NCCREATE)
    if(VBDefControlProc(Handle,Window,WM_NCCREATE,0,lParam))
        {
        Control->Window=Window;
        return TRUE;
```

```
        }
else
    return FALSE;
```

⇓

⇓

另一个需要特殊处理的消息是 WM_NCDESTROY 消息。它标志着控件的消亡。所以也可以用它来通知我们删除它所表示的对象：

⇓

⇓

```
if(Msg==WM_NCDESTROY)
    {
    VbData->Model->Controis--*Control;
    delete Control;
    return VBDefControlProc(Handle,Window,WM_NCDESTROY,0,0);
    }
```

⇓

⇓

剩下的唯一任务就是将针对标准属性的消息分配到默认的处理器中，并且将其它消息传入 model 类中的 Dispatch()函数：

```
switch(Msg)
    {
    case VBM_CHECKPROPERTY:
    case VBM_CETPROPERTY:
    case VBM_GETPROPERTYHSZ:
    case VBM_INITPROPPOPUP:
    case VBM_LOADPROPERTY:
    case VBM_LOADTEXTPROPERTY:
    case VBM_SAVEPROPERTY:
    case VBM_SAVETEXTPROPERTY:
    case VBM_SETPROPERTY:

        if(VbData->Model->Properties[wParam]. IsStandard())
            return VBDefControlProc(Handle,Window,Msg,wParam,lParam);
    }
if(Control)
    return VbData->Model->Dispatch(*Control,Msg,wParam,1Param);
else
    return VBDefControlProc(Handle,Window,Msg,wParam,lParam);
}
```

FindModel()只是简单地遍历 model 对象中的静态列表：

```
model * model::FindModel(HCTL Handle)
  {
  MODEL * SearchModel=VBGetControlModel(Handle);
  for(register unsigned i=0;i<ModelCount;i++)
      if (&ModelList[i]->Model) ==SearchModel)
        return ModelList[i];
  return NULL;
  }
```

Dispatch()方法将消息传入各种虚处理器(virtual hander,即消息处理函数是虚函数),并且在调用虚处理器前,已经将 wParam 和 lParam 参数拆成处理器可以处理的形式。

学习过 MFC 或 ObjectWindows 的人或许会奇怪为什么不象这两个类库那样把 Dispatch()做成虚函数呢?原因就是我讨厌在派生类中使用消息映射之类的宏。如果象 MFC 或 ObjectWindows 那样做,就为每个窗口类维护一个虚方法表(virtual method tables,VMT)。我们很难为 VBX 中的 model 类维护那么多的 VMT,所以我决定将这个函数直接设计成简单的函数:

```
LRESULT model::Dispatch
        (
        control & Control,
        USHORT Msg,
        USHORT wParam,
        long lParam
        )
  {
  switch(Msg)
      {
      case WM_ACTIVATE:
          OnActivate(Control,wParam,
              HIWORD(lParam), (HWND)LOWORD(lParam));
          return 0;
      case WM_CANCELMODE:
          OnCancelMode(Control);
          return 0;
      case WM_CHAR:
          OnChar(Control,wParam,
              LOWORD(lParam), HIWORD(lParam));
          return 0;
      case VBN_CHARTOITEM:
          return OnCharToItem(Control,wParam,
              (HWND)LOWORD(lParam), HIWORD(lParam));
          case WM_CLOSE:
              OnClose(Control);
```

```
            return 0;
//      case VBN COMMAND:
        case VBN COMPAREITEM:
            return OnCompareItem(Control,
                wParam, *((LPCOMPAREITEMSTRUCT)lParam));
        case WM_CREATE:
            return OhCreate(Control, *((LPCREATESTRUCT)lParam));
        case VBN_CTLCOLOR:
            return(LRESULT)OnCuColor(Control,
                devicecontext((HDC)wParam),
                (HWND)LOWORD(lParam), HIWORD(lParam));
        case WM_DEADCIIAR:
            OnDeadChar(Control,wParam,
                LOWORD(lParam), HIWORD(lParam));
            retun 0;
        case VBN_DELETEITEM:
            OnDeleteItem(Control,wParam,
                 *((LPDELETEITEMSTRUCT)lParam));
            return 0;
        case WM_DESTROY:
            OnDestroy(Control);
            return 0;
        case VBN_DRAWITEM:
            OnDrawItem(Control,wParam,
                *((LPDRAWITEMSTRUCT)lParam));
            return 0;
        case WM_DROPFILES:
            OnDropFiles(Control,(IIANDLE)wParam);
            return 0;
        case WM_ENABLE:
            OnEnable(Control,wParam);
            return 0;
        case WM_ERASEBKGND:
            OnEraseBkgnd((Control,devicecontext(HDC)wParam));
            return 0;
        case WM_GETFONT:
            return(LRESULT)OnGetFont(Control);
        case WM_GETMINMAXINFO:
            OnGetMinMaxInfo(Control, *((MINMAXINFO far*)lParam));
            return 0;
```

```
case WM_GETTEXT:
    OnGetText(Control,wParam(LPSTR)lParam));
    return 0;
case WM_GETTEXTLENGTH:
    return OnGetTextLength(Control);
case WM_HSCROLL:
case VBN_HSCROLL:
    OnHScroll(Control,wParam,
        LOWORD(lParam), (HWND)HIWORD(lParam));
    return 0;
case WM_KEYDOWN:
    OnKeyDown(Control,wParam,
        LOWORD(lParam), HIWORD(lParam));
    return 0;
case WM_KEYUP:
    OnKeyUp(Control,wParam,
        LOWORD(lParam), HIWORD(lParam));
    return 0;
case WM_KILLFOCUS:
    OnKillFocus(Control,(HWND)wParam);
    return 0;
case WM_LBUTTONDBLCLK:
    OnLButtonDblClk(Control,wParam,point(lParam));
    return 0;
case WM_LBUTTONDOWN:
    OnLButtonDown(Control,wParam,point(lParam));
    return 0;
case WM_LBUTTONUP:
    OnLButtonUp(Control,wParam,point(lParam));
    return 0;
case WM_MBUTTONDBLCLK:
    OnMButtonDblClk(Control,wParam,point(lParam));
    return 0;
case WM_MBUTTONDOWN:
    OnMButtonDown(Control,wParam,point(lParam));
    return 0;
case WM_MBUTTONUP:
    OnMButtonUp(Control,wParam,point(lParam));
    return 0;
case VBN_MEASUREITEM:
```

```
        OnMeasureItem(Control,wParam,
            *((LPMEASUREITEMSTRUCT)lParam));
        return 0;
    case WM_MOUSEMOVE:
        OnMouseMove(Control,wParam,point(lParam));
        return 0;
    case WM_MOVE:
        OnMove(Control,point(lParam));
        return 0;
    case WM_NCHITTEST:
        return OnNCHitTest(Control,point(lParam));
    case WM_PAINT:
        OnPaint(Control);
        return 0;
    case WM_POWER:
        return OnPower(Control,wParam);
    case WM_QUERYENDSESSION:
        return OnQueryEndSession(Control);
    case WM_RBUTTONDBLCLK:
        OnRButtonDblClk(Control,wParam,point(lParam));
        return 0;
    case WM_RBUTTONDOWN:
        OnRButtonDown(Control,wParam,point(lParam));
        return 0;
    case WM_RBUTTONUP:
        OnRButtonUp(Control,wParam,point(lParam));
        return 0;
    case WM_SETCURSOR:
        return OnSetCursor(Control(HWND)wParam,
            LOWORD(lParam), HIWORD(lParam));
    case WM_SETFOCUS:
        OnSetFocus(Control,(HWND)wParam);
        return 0;
    case WM_SETFONT:
        OnSetFont(Control,(HFONT)wParam, lParam != 0);
        return 0;
    case WM_SETTEXT:
        OnSetText(Control,(LPSTR)lParam);
        return 0;
    case WM_SIZE:
```

```
        OnSize(Control,size(lParam));
        return 0;
    case WM_SPOOLERSTATUS:
        OnSpoolerStatus(Control,wParam,(UINT)lParam);
        return 0;
    case WM_SYSCOLORCHANGE:
        OnSysColorChange(Control);
        return 0;
    case WM_TIMECHANGE:
        OnTimeChange(Control);
        return 0;
    case WM_TIMER:
        OnTimer(Control,wParam);
        return 0;
    case VBN_VKEYTOITEM:
        return OnVKeyToItem(Control,wParam,
            (HWND)LOWORD(lParam), HIWORD(lParam));
    case WM_VSCROLL;
    case VBN_VSCROLL:
        OnVScroll(Control,wParam,
            LOWORD(lParam), (HWND)HIWORD(lParam));
        return o;
    case WM_WININICIIANGE:
        OnWinIniChange(Control,(LPSTR)lParam);
        return 0;
//  case VBM_LOADED:
    case VBM_CANCELMODE:
        VbCancelMode(Control,(HWND)wParam);
        return 0;
    case VBM_CHECKPROPERTY:
        return Properties[wParam]. TestValue(Control,lParam);
//  case VBM_COPY:
//  case VBM_CREATED:
//  case VBM_DATA_AVAILABLE:
//  case VBM_DATA_GET:
//  case VBM_DATA_INITLATE:
//  case VBM_DATA_METHOD:
//  case VBM_DATA_REQUEST:
//  case VBM_DATA_SET:
//  case VBM_DATA_TERMINATE:
```

```
//      case VBM_DRAGDROP:
//      case VBM_DRAGOVER:
//      case VBM_FIREEVENT:
//      case VBM_GETDEFSIZE:
//      case VBM_GETPALETTE:
//      case VBM_GETPROPERTY:
            return Properties[wParam]. GetValue(Control,lParam);
//      case VBM_GETPROPERTYHSZ:
//      case VBM_HELP:
//      case VBM_HITTEST:
//      case VBM_INITPROPPOPUP:
//      case VBM_ISMNEMONIC:
//      case VBM_LINKENUMFORMATS:
//      case VBM_LINKGETDATA:
//      case VBM_LINKGETITEMNAME:
//      case VBM_LINKSETDATA:
//      case VBM_LOADPROPERTY:
            return Properties[wParam]. LoadValue(Control,
                (HFORMFILE)lParam);
//      case VBM_LOADTEXTPROPERTY:
        case VBM_METHOD:
            return OnMethod(Control,wParam,lParam);
//      case VBM_MNEMONIC:
        case VBM_PAINT:
            VbPaint(Control,(HDC)wParam, (LPRECT)lParam);
        return 0;
//      case VBM_PAINTMULTISEL:
//      case VBM_PAINTOUTLINE:
//      case VBM_PALETTECHANGED:
//      case VBM_PASTE:
//      case VBM_QPASTEOK:
//      case VBM_SAVEPROPERTY:
            return Properties[wParam]SaveValue(Control,
                (HFORMFILE)lParam);
//      case VBM_SAVETEXTPROPERTY:
//      case VBM_SELECTED:
//      case VBM_SETPROPERTY:
            return Properties[wParam]. SetValue(Control,lParam);
//      case VBM_WANTSPECIALKEY:
        }
```

```
return VBDefControlProc(Control,Control,Msg,wParam,lParam);
}
```

你一定注意到了，许多消息标识符被注释掉了。我没有为它们提供消息处理器。我只是将这些深奥的消息留到我需要的时候再实现它们。所以我在这里没有处理它们。

消息处理器都非常简单，所以我就不在这里列举它们了。我们只来看看一个简单的例子：

```
LRESULT model::OnInitialize(control & Control)
  {
  for(register UINT p=0;p<Properties.GetCount()p++)
       for(register i=0;i<Properties[p].GetElementCount(); i++)
            if(!Properties[p].IsStandard())
                 Properties[p].SetInitValue(Control,i);
  return VBDefControlProc(Control,Control,VBM_INITIALIZE,0,0);
  }
```

默认的消息处理器都简单地将消息传入 VBDefControlProc()函数中。control(控件)类已经进行了 HCTL 和 HWND 类型强制转换，所以在调用 VBDefControlProc()函数时就不必显式地进行类型转换了。

最后，我还有一点需要说明：就是 OnPaint()、VbPaint()和 OnDraw()这三个消息处理器。前两个处理器是真正的相应消息的处理器，但它们不是虚函数。那是因为程序员不应该重载它们。OnPaint()只是生成一个 paintcontext 对象并且以它为参数调用虚函数 OnDraw()：

```
void model::OnPaint(control & Control)
  {
  paintcontext Dc(Control);
  OnDraw(Control,Dc,Dc.ShouldRedrawBackground(), Dc.GetClipRect());
  }
```

VbPaint()函数的操作是简单的。它要依赖于一个设备环境对象。实际上，这里千万不要调用 BeginPaint()函数，因为当一个控件需要在其它设备而不是屏幕上作图时，VBM_PAINT 消息会发送给它；此外对于图形控件来说，它只接收 VBM_PAINT 消息，而不接收 WM_PAINT 消息：

```
void model::VbPaint(control & Control,HDC dc,LPRECT pRect)
  {
  devicecontext Dc(dc);
  OnDraw(Control,Dc,TRUE,rectangle(*pRect));
  }
```

默认的 OnDraw()函数只是画一个控件所占区域的矩形。如果控件是图形控件，这个函数画一个绿色矩形；否则它画一个蓝色矩形：

```
void model::OnDraw(control &,devicecontext & Dc,BOOL,rectangle Rect)
  {
  BOOL IsGrapincal=((Model.fl & MODEL_fGraphical)==MODEL_fGraphical);
  brush Brush(IsGraphical?color::Green:color:Blue);
  Dc=Brush;
```

```
Dc.DrawRectangle(Rect);
}
```

注意到了将新恻子选择到设备环境中去的赋值操作符吗?它看起来是不是很自然?

6.3 处理方法

所有的方法都是通过单独的 VBM_METHOD 消息来调用的。这个消息的参数标志着哪个方法要被调用,而其它参数的意义则依赖于各个方法。这个消息的处理器是 OnMethod(),一个非虚函数:

```
LRESULT model::OnMethod(control & Control,WORD MethodID,LONG Args)
  {
  LRESULT Result=0;

  switch(MethodID)
      {
      case METH_ADDITEM:
          {
          LPADDITEM AddItem=(LPADDITEM)Args;
          LPSTR Item=VBLockHsz(AddItem->Item);
          int Index=(AddItem->Count>2)?(int)AddItem->Index: -1;
          Result=OnMethodAddItem(Control,Item,Index);
          VBUnlockHsz(AddItem->Item);
          }
          break;
  case METH_CLEAR:
      Result=OnMethodClear(Control);
      break;
  case METH_DRAG:
      {
      LPDRAG Drag=(LPDRAG)Args;
      int Command=(Drag->Count>1)?(int)Drag->Command: 1;
      Result=OnMethodDrag(Control,Command);
      }
      break;
  case METH_LINKSEND:
      Result=OnMethodLinkSend(Control);
      break;
  case METH_MOVE:
      {
      LPMOVE Move=(LPMOVE)Args;
```

```
    Result-OnMethodMove(Control,
        (int)Move->Count, Move->Left, Move->Top,
        Move->Width, Move->Height);
    }
    break;
case METH_REMOVEITEM:
    {
    LPREMOVEITEM RemoveItem=(LPREMOVEITEM)Args;
    Result-OnMethodRemoveltern(Control,
        (int)RemoveItem->Index);
    }
    break;
case METH_REFRESH:
    Result=OnMethodRefresh(Control);
    break;
case METH_ZORDER:
    {
    LPZORDER ZOrder-(LPZORDER)Args;
    Result=OnMethodZOrder(Control,
        ZOrder->SendToBack?SendToBack:BringToFront);
    }
    break;
default:
    Result=VBDefControlProc(Control,Control,
        VBM_METHOD,MethodID,(long)Args);
    }
    return Result;
    }
```

这个消息开关识别不同的方法，适当地重新拆分参数并调用与那个方法相应的虚函数。

当然，下面这些函数就是当你创建一个实际的需要使用这些方法的控件时你所要重载的函数。这些基本的实现提供了恰当的默认处理。通常的默认处理是简单地调用 VBDefControlProc()。由于我们拆分了参数，所以如果要进行那样的默认处理，我们就必须为消息处理器重新构造参数结构。此外，一些方法的默认处理只是简单地返回错误代码444，即不能处理这个方法：

```
LRESULT model::OnMethodAddItem(control &,LPSTR,int)
    {
    return 444;
    }
LRESULT model::OnMethodClear(control &)
    {
    return 444;
```

```
    }
  LRESULT model::OnMethodRemoveItem(control &,int)
    {
    return 444;
    }
```
另外一些方法的处理中很容易调用 VBDefControlProc()，因为它们并不需要什么参数：

```
  LRESULT model::OnMethodLinkSend(control & Control)
    {
    return VBDefControlProc(Control,Control,
         VBM_METHOD,METH_LINKSEND,NULL);
    }
  LRESULT model::OnMethodRefresh(control & Control)
    {
    return VBDefControlProc(Control,Control,
         VBM_METHOD,METH_REFRESH,NULL);
    }
```
那只剩下两个方法的确需要我们重新构建最初参数后再调用 VBDefControlProc()了：

```
LRESULT model::OnMethodMove(control & Control,
            int Count,LONG Len,LONG Top,LONG Width,LONG Height)
    {
    MOVE Args;
    Args.Count=Count;
    Args.Left=Left;
    Args.Top=Top;
    Args.Width=Width;
    Args.Height=Height;
    return VBDefControlProc(Control,Control,
         VBM_METHOD,METH_MOVE,(long)& Args);
    }
LRESULT model::OnMethodZOrder(control & Control,ZOrderCommand Command)
   {
   ZORDER Args;
   Args.Count=2;
   Args.SendToBack=(Command==SendToBack);
   return VBDefControlProc(Control,Control,
        VBM_METHOD,METH_ZORDER,(long)& Args);
   }
```

6.4 control(控件)类

这个类封装控件和运行中的控件窗口的句柄。它包含有指向每个控件复本中属性存贮区

的指针，并且包含有转换到控件句柄和窗口句柄的强制类型转换操作符。

正如我们前面看到的，control 类对象是在响应 VBM_INITIALIZE 消息时生成的，并且当接收到 WM_NCCREATE 消息时要填写 Window 属性值。(图形控件不会接收到这个消息，它的 Window 属性为 NULL。)control 对象是在响应 WM_NCDESTROY 时被删除的，即使是图形控件也会接收到这个消息。

control 类是在 CONTROL.H 中说明、在 CONTROL.CPP 中定义的：

```
class control
  {
  public:
      control(HCTL aHandle,model*aModel);
      int operator==(const control & aControl)const;
      void*GetStorageAddress(int Offset)const;
      operator HCTL(void)const {return Handle;}
      operator HWND(void)const {return Window;}
      model & GetModel(void)const {return*Model;}
    private:
          HCTL Handle;
          HWND Window;
          model*Model;
 friend LRESULT CALLBACK_export CtlProc(HCTL,HWND,
      USHORT,USHORT,long);
 }
typedef control*control_ptr;
```

注意，CtlProc()是 model 类的友元函数，同时也是 control 类的友元函数。如果需要的话，control 类对象很容易被强制转换成 HWND 或 HCTL 类型。这里我并不想呈现构造函数和等价操作符的定义，因为它们的实现实在是平淡无奇。下面是定义在 CONTROL.CPP 文件中的 GetStorageAddress()函数的实现代码：

```
void*control::GetStorageAddress(int Offset)const
  {
  return(void*)((char*)VBDerefControl(Handle)+Offset);
  }
```

给定一个偏移量(Offset)，这个函数就可以获取 VBDATA 结构的地址，再加上这个偏移量并返回相加的结果。我们很快就会在后面见到它的用法。

我们还设计了 controlbag 类，因为我们需要一个非排序的容器，并且 ObjectWindows 容器依赖于模板来建造(Visual C++1.52不支持模板)，而 MFC3.0根本不支持袋类(bag-type)容器。此外，这个类本身是非常简单的：

```
class controlbag
  {
  public:
      controlbag(UINT aGrowSize=50);
```

```
        virtual~controlbag();
        controlbag & operator+=(control & Control);
        controlbag & operator-=(control & Control);
        control* Find(HCTL Handle);

    private:
        void Grow(void);
        UINT GrowSize;
        control_ptr* Controls;
        UINT ControisMax;
        UINT ControlsCount;
```

“袋子”被保存在动态维护的指针数组中，并且这个数组在构造函数中被分配，而在析构函数中被释放掉：

```
controlbag::controlbag(UINT aGrowSize)
    GrowSize(aGrowSize),
    ControlsMax(aGrowSize),
    ControlsCount(0)
  {
  Controls=new control_ptr[GrowSize];
  }
controlbag::~controlbag()
  {
   for(register UINTi=0;i<ControlsCount;i++)
      delete Controls[i];
   delete[]Controls;
  }
```

如果加入的控件数量大于袋子的空间，袋子要被重新分配：

```
void controlbag::Grow(void)
  {
  control_ptr* Temp=Controls;
  ControlsMax+=GrowSize;
  Controls=new control_ptr[ControlsMax];
  for(register UINTi=0;i<ControlsCount;i++)
      Controls[i]=Temp[i];
  delete[]Temp;
  }
```

我不是使用“add(加入)”方法来向袋中加入控件的，而是使用了加法赋值操作符：

```
controlbag & controlbag::operator+=(control & Control)
  {
```

```
    if(ControlsCount==ControlsMax)
        Grow();
    Controis[ControisCount++]=&Control;
    return*this;
    }
```

此外，减法赋值操作符则用于删除袋中的元素：

```
controlbag & controlbag::operator-=(control & Control)
    {
    for(register UINTi=0;i<ControlsCount;i++)
        if(HCTL)*Controls[i]==(HCTL)Control)
            {
            Controls[i]=Controls[--ControlsCount];
            break;
            }
    return*this;
    }
```

(被删除的元素在这里并没有被 delete 掉，因为它是由参数传入的，并且应由调用这个函数的对象来负责。)

最后，Find()方法被 CtlProc()函数用来根据控件句柄来定位控件对象：

```
control*controlbag::Find(HCTL Handle)
    {
    for(register UINTi=0;i<ControlsCount;i++)
        if(HCTL)*Controls[i]==Handle)
            return Controls[i];
    return NULL;
    }
```

6.5 Property(属性)类

是什么使 VBX 风格的用户控件变得很独特?有三点：它的外观、它的事件列表和它的属性。属性类支持属性名和属性存贮，这使你很容易在你的控件中建造任何标准属性或用户属性。

正如你知道的，一些 VBM_消息影响属性的值。对于标准属性，这些消息必须进行默认的处理。当消息到达控件过程(前面已讲过，控件过程是 model 类的一部分)时，wParam 参数包含有属性的索引。如果是标准属性，property::IsStandard()将函数返回 TRUE；针对非标准属性的消息直接传递给消息发送器。我们在前面已见过，model::Dispatch()函数将消息参数进行拆分并对每个消息调用虚处理器。大多数消息处理器都是 model 类的方法，但处理属性值的消息处理器由 customproperty 对象来管理。

standardproperty 类是属性类的基类，它说明在 PROPS.H 中。它保存标准控件的 PROPINFO 值，并且提供询问信息的虚函数：IsStandard()和 GetStorageSize()：

```
class standardproperty
  {
  public:
       standardproperty (PPROPINFO aStdProperty,USHORT aVersion=
       VB100_VERSION);
       virtual BOOL IsStandard(void)const;
       virtual UINT GetStorageSize(void)const;
  protected:
       enum errorvalues
            {
            InvalidValue=380,
            InvalidIndex=381
            }
       private:
            PPROPINFO PropertyID;
            USHORT Version;
       friend class propertieslist;
       };
```

这个类主要的任务是充当 customproperty 的基类。但它自身也是非常有用的，因为标准属性是用它的 ID 来表示的。这个类中 ID 被存贮在 PropertyID 域中。同时你还要留意一下错误返回值的匿名枚举变量。

因为这个类的功能很单一，所以在 PROPS. CPP 中的那三个方法：构造函数、IsStandard()和 GetStorageSize()的实现是非常简单的：

```
standardproperty::standardproperty(PPROPINFO aStdProperty,USHORT aVersion)
     PropertyID(aStdProperty),
       Version(aVersion)
  {
  }
BOOL standardproperty::IsStandard(void)const
  {
  return TRUE;
  }
UINT standardproperty::GetStorageSize(void)const
  {
  return 0;
  }
```

派生类 customproperty 则要复杂得多，尽管你根本不会直接使用它。还记得 CDK 的 PROPINFO 结构的那些域吗？这个类必须负责管理它们，并且当它们被激活时处理这些属性，比如存贮和装入等等：

```
class customproperty: public standardproperty
```

画笔是通过使用赋值操作符被选入 devicecontext 对象的，接下来就是画线。这时代码已经到了 for 语句的结尾处，因此 pen 对象结束了它的生命周期。在画笔对象被撤消之前，如果画笔被选入设备环境中，画笔的析构函数会自动从设备环境中删除它。

最后，我们需要通过生成 gradient 类的一个实例来促使以上这些操作的发生：

gradient Gradient；

这行语句给了我们所要的一切：GRADIENT.VBX 诞生了！我们可以把它加入到一个工程的 Visual Basic 控件列表中；我们可以把它放在一个表中；我们不但可以设置 TopColor 的值，也可以设置它在表中的位置。图6.4就是一个例子。

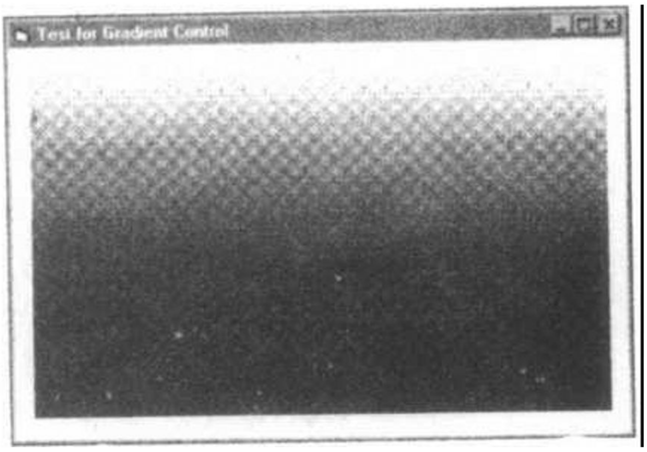

图6.4 Gradient 控件使用的例子

```
        PROPINFO PropertyInfo;
        BYTE ElementCount;
        BYTE ElementSize;
        BOOL RefreshOnChange;
    private:
        BOOL IsArray(void)const;
        UINT StorageOffset;
    };
```

第一个数据项是枚举型的 datatypes，它提供了比 CDK 更易读懂的 VB 数据类型的列表。注意：这个列表并没有列出全部的 VB 数据类型。我去掉了 VB 字符串类型(HLSTR)，因为这个数据类型对我们没有多大用途。如果你需要文本属性，C 风格字符串就足够了。并且我还去掉了 DT_OBJECT 类型，因为这种类型用 OCX 实现会更好。

类的成员数据放在了说明的底部。其中 ElementSize 在派生类中用来规定所需存贮空间大小。ElementCount 成员表明这个类中所有的属性都是按属性数组来处理的！这并不是考虑到 VB 的习惯，而是从你的用户属性(customproperty)派生类角度出发的。相信我，这样做会使事情变得更容易。

现在，让我们看看该类的构造函数：

```
customproperty::customproperty(LPSTR aName,datatypes aDataType)
        standardproperty((PPROPINFO)&propertyInfo, 0),
        StorageOffset(0),
        ElementCount(1),
        ElementSize(0),
       RefreshOnChange(TRUE)
  {
  lstrcpy(Name,aName);
  PropertyInfo.npszName=(PSTR)Name;
  PropertyInfo.fl=(FLONG)aDataType|
      PF_fGetMsg|PF_fSetMsg|PF_fSaveMsg;
  PropertyInfo.offsetData=0;
  PropertyInfo.infoData=0;
  PropertyInfo.dataDefault=0;
  PropertyInfo.npszEnumList=NULL;
  }
```

你可能还记得，标准属性的 PROPINFO 值根本不是指针，而是带有位识别操作的索引。底层 VBX 机制通过这个索引就可以准确无误地找到它所对应的指针。实际上，它就存贮在 PropertyID 成员中。用户控件必须在构造函数初始化成员时将真正的 PROPINFO 结构的地址赋值给它。接下来，构造函数要填写这个结构的域值。

standardproperty 类中的 IsStandard()虚函数返回 TRUE。而 customproperty 类中的这个函数必须返回 FALSE：

```
BOOL customproperty::IsStandard(void)const
```

```
{
return FALSE;
}
```

接下来是几个存取成员数据的监视函数。SetAsArray()被设计成内嵌函数，它把用户属性的外观呈交给 VBX 管理器来处理。它用来设置一组元素的值，包括加入 PF_fPropArray 标志：

```
inline void customproperty::SetAsArray(UINT anElementCount)
  {
  ElementCount=anElementCount;
  PropertyInfo.fl|=PF_fPropArray;
  }
```

既使你的派生类自己管理动态数组的存贮空间，你可能仍需要调用这个函数，因为你规定的元素个数在后面要用来确保数组索引的有效性。

SetStorageOffset()也是内嵌函数。它是当属性的存贮空间大小被确定时，由 model 对象来调用的(这与把一个入口放在我们前面介绍的 VBX 框架的 VBDATA 结构中是一样的)：

```
inline void customproperty::SetStorageOffset(int anOffset)
  {
  StorageOffset=anOffset;
  }
```

GetStorageOffset()要使用索引参数：

```
inline int customproperty::GetStorageOffset(LONG Index)const
  {
  return StorageOffset+(ElementSize*(int)Index);
  }
```

记住，既使是非数组类属性，这里也是按照单元素的数组处理的。这种情况下 Index 通常是0。

GetStorageSize()是一个虚函数，它根据 ElementSize 和 ElementCount 来计算所需要的存贮空间的大小：

```
UINT customproperty::GetStorageSize(void)const
  {
  return ElementSize*ElementCount;
  }
```

SetRetreshOnChange()方法是一 个内嵌函数，负责通知对象什么时候 VBX 管理器改变了这个属性值，控件应该重画自身。

接下来的三个方法是由 model::Dispatch()来调用的，目的是让 VBX 管理器通知控件测试、设置或获取某个属性值。这些方法又调用了与自己同名的重载方法。而这些重载方法都是虚的，所以派生类确实能接收到消息。让我们来看看第一个消息处理器 TestMessage()，它是在响应 VBM_CHECKPROPERTY 消息时调用的，并且它需要调用另一个虚方法 TestIndex()：

```
LRESULT customproperty::TestIndex(LONG Index)const
```

```
    {
    return(Index>=0 && Index<ElementCount)?0: InvalidIndex;
    }
LRESULT customproperty::TestValue(control & Control,LONG Value)const
    {
    LPDATASTRUCT DataStruct((LPDATASTRUCT)Value);
    LONG Index(0);
    if(IsArray())
        {
        Index=DataStruct->index [0] data;
        Value=DataStruct->data;
        }
    LRESULT Error(TestIndex(Index));
    return Error?Erro: TestValue(Control,Value,Index);
    }
```

我们知道，在 CDK 中数组属性的处理与非数组属性的处理是不同的。对于数组属性来说，lParam 参数(这里叫 Value)是指向 DATASTRUCT 的指针：所以这个结构的 Data 属性既可以是数据也可以是数据的指针，这依赖于数据的类型。

为了避免分别单独处理数组和非数组属性，所有的 customproperty 对象把它们都看作是数组，即使有的数组只含有一个元素。而且这并不影响 VBX 管理器对它们的处理。所以第一层次的 TestValue()函数的任务是确保 Value 含有正确的值，再设置 Index 变量并调用下一个层次的 TestValue()。

设置 Index 变量是使 TestValue()能检验索引是否有效。一般情况下，customproperty 类提供的虚函数 TestIndex()可以完成这个任务。很少会有这种情况：你使用越界索引来表示特殊意义，如 Listbox(列表框)控件使用－1索引来表示列表框中没有表项。你可以在派生类中重载 TestIndex()函数。

VBM_SETPROPERTY 消息在语法上与 VBM_CHECKPROPERTY 消息有些相似，但其目的是改变属性的值：

```
LRESULT customproperty::SetValue(control & Control,LONG Value)
    {
    LPDATASTRUCT DataStruct(LPDATASTRUCT)Value);
    LONG Index(0);
    if(IsArray())
        {
        Index=DataStruct->index [0] data;
        Value=DataStruct->data;
        }
    LRESULT Error(TestIndex(Index));
    if(!Error)
        Error=SetValue(Control,Value,Index);
```

```
if((!Error)&& RerfreshOnChange)
    VBInvalidateRect(Control,NULL,FALSE);
return Error;
}
```

因为并不能保证 VBM_CHECKPROPERTY 消息一定会在 VBM_SETPROPERTY 消息之前发送，所以一定要对索引值重新进行测试。如果派生类的 SetValue()函数不返回错误并且属性值的改变应导致控件的重画，就要调用 VBInvalidateRect()确保重画控件。

VBM_GETPROPERTY 消息的处理有一点不同，lParam 通常是指向所需数据的指针。但如果这个属性是一个数组，DATASTRUCT 的 Data 域通常不是一个指针，而是数组本身：

```
LRESULT customproperty::GetValue(control & Control,LONG lParam)const
  {
  LPDATASTRUCT DataStruct((LPDATASTRUCT)lParam);
  LONG far*Data=((LONG far*)lParam);
  LONG Index(0);
  if(IsArray())
      {
      Index=Data Stwct->index [0] .data;
      Data=&DataStruct->data;
      }
  LRESULT Error(TestIndex(Index));
  return Error: Error: GetValue(Control,*Data,Index);
  }
```

这里所调用的 GetValue()方法的第二个参数是一个引用类型：Data 本身是一个指针，所以既可以是 lParam 所指定的指针值，也可以指向 DataStruct 结构的 data 域。

以上那些函数所调用的 SetValue()和 GetValue()函数都是虚函数。因为有几个属性数据类型不允许在数值测试时失败(比如 long 型)。所以针对这些数据类型有一个默认的什么都不做的 TestValue()函数。同样，为了方便，有必要为 SetValue()和 GetValue()提供什么都不做的默认处理的版本：

```
LRESULT customproperty::TestValue(control &,LONG,LONG)const
  {
  return 0;
  }

LRESULT customproperty::SetValue(control &,LONG,LONG)
  {
  return 0;
  }

LRESULT customproperty::GetValue(control &,LONG&,LONG)const
  {
```

```
return 0;
}
```

下面的两个方法是在响应 VBM_LOADPROPERTY 消息和 VBM_SAVEPROPERTY 消息时分别被调用的。在这里它们是虚函数，但一般不应该重载它们：

```
LRESULT customproperty::LoadValue(control & Control,HFORMFILE File)
{
void * Value=Control.GetStorageAddress(GetStorageOffset());
return VBReadFormFile(File,Value,ElementSize * ElementCount);
}
LRESULT customproperty::SaveValue(control & Control,HFORMFILE File)const
{
void * Value=Control.GetStorageAddress(GetStorageOffset());
return VBWriteFormFile(File,Value,ElementSize * ElementCount);
}
```

最后，我们要处理属性的默认值。VBX CDK 允许通过在 PROPINFO 结构中设置 PF_fDefVal 标志来指定默认的属性值。但这只是告诉 VBX 管理器默认的属性是什么，并不真正存贮属性的值。也就是说，它并不真正设置数值。我们可以克服这一点。首先跳回到 model 类，我们可以在 On Initialize()处理器中设置初始的值：

```
LRESULT model::OnInitialize(control & Control)
{
for(register UINT p=0;p<Properties.GetCount(); p++)
    for(register i=0;i<Properties[p]GetElementCount(); i++)
        if(!Properties[p].sStandard())
    Properties [p] .SetInitValue(Control,i);
return VBDefControlProc(Control,Control,VBM_INITIALIZE,0,0);
}
```

正如你所看到的，每个属性的每个元素都被初始化了。但它们是如何被初始化的?这是 SetInitValue()方法的任务：

```
void customproperty::SetInitValue(control &Control,LONG Index)
{
if((PropertyInfo.fl & PF_fDefVal)==PF_fDefVal)
    SetValue(Control,PropertyInfo.dataDefault,Index);
}
```

它检查 PF_fDefVal 标志是否被设置。通常是没有设置，这就什么都不会发生。但如果设置了，PropertyInfo.dataDefault 域中包含有所需要的值，并且这个域正好是一个 **long** 类型。所以调用前面所讲过的 SetValue()方法就可以完成任务了。

我们后面所讲到 colorproperty(颜色属性)类时就真正使用了这个机制。

6.5.1 boolproperty(布尔属性)类

我们没有足够的篇幅来讲解所有的 customproperty 的派生类，我只能通过讲解其中的几

个类来使你了解这些类是如何设计的。第一个数据类型是Boolean(布尔型)，所以我们从这个类开始。下面是这个类的说明：

```
class boolproperty: public customproperty
  {
  public;
     boolproperty(LPSTR aName);
     virtual LRESULT TestValue(control & Control,LONG Value,LONG Index)const;
     virtual LRESULT SetValue(control & Control,LONG Value,LONG Index);
     virtual LRESULT GetValue(control & Control,LONG& Value,LONG Index)const;
     BOOL GetValue(control & Control,LONG Index=0)const;
  }
```

这个类只包含有一个构造函数和三个方法。customproperty 的这些派生类大多是这样的。下面是这个类的构造函数：

```
boolproperty::boolproperty(LPSTR aName)
  :     customproperty(aName,Boolean)
  {
  ElementSize=sizeof(BYTE);
  }
```

我们打算用单个字节存贮布尔值，所以构造函数设置ElementSize为一个BYTE大小。此外，在调用基类的构造函数时设置datatype为Boolean。

虽然谁也不会对布尔类型的属性设置除了TRUE(真)和FALSE(假)以外其它的值，但我们还是应该检查一下属性值的有效性：

```
LRESULT boolproperty::TestValue(control &,LONG Value,LONG)const
  {
  if(Value==-1||Value==0)
     return 0;
  else
      return InvalidValue;
  }
```

虽然这个方法不会被直接调用(除非基于boolproperty再派生一个类并设置PF_fSetCheck标志，生成这个类对象时会直接调用这个方法)，但我们在SetValue()函数中调用了它，只是为了确保属性值的有效性：

```
LPESULT boolproperty::SetValue(control & Control,LONG Value,LONG Index)
  {
  LRESULT Error=TestValue(Control,Value,Index);
  if(!Error)
     {
     BYTE* Storage=
         (BYTE*)Control.GetStorageAddress(GetStorageOffset(Index));
```

```
        *Storage=((BYTE)Value!=0);
        }
    return Error;
    }
```

现在你应该明白为什么要有两个层次的 SetValue()(以及其它函数)了吧：虽然 customproperty 层次知道如何处理 lParam 和 DATASTRUCT 结构，但它的派生类层次仍需知道如何将结构转换成正确的数据类型。这里对于布尔类型的数据来说，转换是非常简单的。后面我们就会看到，对于 Picture、Float 和 CString 类型来说，转换要复杂得多。

GetValue()相对简单一些，是因为获取一个数值一般不会失败(只要给定的数组索引是有效的)：

```
LRESULT boolproperty::GetValue(control & Control,LONG & Value,LONG Index)con-
    st
    {
    BYTE* Storage=
        (BYTE*)Control.GetStorageAddress(GetStorageOffset(Index));
    Value=*Storage?-1: 0;
    return 0;
    }
```

6.5.2 colorproperty(颜色属性)类

colorproperty 类允许规定默认值，这是构造函数中处理的：

```
colorproperty::colorproperty(LPSTR aName,color DefColor)
    :       customproperty(aName,Color)
    {
    ElementSize=sizeof(COLORREF);
    PropertyInfo.dataDefault=(COLORREF)DefColor;
    PropertyInfo.fl|=PF_fDefVal;
    }
```

除了象所有的 customproperty 派生类必须要设置 ElementSize 值外，colorproperty 类还设置了 RefreshOnChange 标志(这样，如果颜色改变了，控件就必须要重画)。它也接受可选的 DefColor 参数(默认的颜色是 color::Black)并设置了 PF_fDefVal 标志。

6.5.3 enumproperty(枚举属性)类

下一个有趣的类是 enumproperty(枚举属性)类：

```
class enumproperty: public customproperty
    {
    public:
        enumproperty(LPSTR aName,LPSTR anEnumList);
        virtual LRESULT TestValue(control & Control,LONG Value,LONG Index)const;
        virtual LRESULT SetValue(control & Control,LONG Value,LONG Index);
```

```
    virtual LRESULT GetValue(control & Control,LONG & Value,LONG Index)con-
    st;
    UINT GetValue(control & Control,LONG Index=0)const;
  private:
    BYTE EnumCount;
    char EnumList[256];
  }
```

它的构造函数必须接受一个枚举类值的列表：

```
enumproperty::enumproperty(LPSTR aName,LPSTR anEnumList)
    customproperty(aName,Enumerated),
    EnumCount(0)
  {
  ElementSize=sizeof(BYTE);
  PropertyInfo.npszEnumList=(PSTR)EnumList;
  register i(1);
  if(anEnumList)
      {
      EnumList[0]—anEnumList[0]
      while (anEnumList[i-1]||anEnumList[i])
          {
          EnumList[i]=anEnumList[i];
          if(anEnumList[i++]==0)
              EnumCount++;
          }
      }
  PropertyInfo.enurnMax=EnumCount;
  }
```

这个列表使用的是标准 CDK 格式(即"Choice 1\0Choice 2\0Choice 3\0",一个以双 NULL 结尾的字符串,其中一个空字符是编译器加上的)。对象必须扫描它以便知道一共有多少个选项。这个值不但用于设置 PROPERTYINFO. enumMax 域，而且用于检验对这个属性的赋值：

```
LRESULT enumproperty::TestValue(control &,LONG Value,LONG)const
  {
  LRESULT Error(0);
  if(Value<0||Value>=EnumCount)
      Error=InvalidValue;
  return Error;
  }
```

SetValue()和 GetValue()这两个方法与 booleanproperty 类中相应的函数是相似的：

```
LRESULT enumproperty::SetValue(control & Control,LONG Value,LONG Index)
  {
```

```
BYTE* Storage=(BYTE* )Control. GetStorageAddress(GetStorageOffset(Index));
*Storage=(BYTE)Value;
return 0;
}
LRESULT enumproperty::GetValue(control & Control,LONG & Value,LONG Index)
 const
 {
 BYTE* Storage=(BYTE* )Control. GetStorageAddress(GetStorageOffset(Index));
 Value=*Storage;
 return 0;
 }
```

6.5.4 textproperty(文本属性)类

实际上存贮 C 风格的字符串是一件令人头痛的事情，因为属性类数据项要覆盖的内存空间是不定的。这就是为什么要设计 text 类(在第五章)的原因。作为能存贮甚至处理文本字符串的与软件商无关的类，要能在内部管理字符串的存贮，我们所存贮的就是对象本身。当然，由于我们实际上应该提供 LoadValue()和 SaveValue()重载函数，这两个函数中的从磁盘装入属性和将属性存入磁盘的操作要稍微复杂一些：

```
class textproperty: public customproperty
 {
 public:
     textproperty(LPSTR aName);
     virtual LRESULT SetValue(control & Control,LONG Value,LONG Index);
     virtual LRESULT GetValue(control & Control,LONG& Value,LONG Index)con-
     st;
     text & GetValue(control & Control,LONG Index=0)const;
     virtual LRESULT LoadValue(control & Control,HFORMFILE File);
     virtual LRESULT SaveValue(control & Control,HFORMFILE File)const;
 };
```

由于构造函数必须要确定属性的(线性)存贮空间的大小(这个空间与要存贮的字符串的实际大小是无关的)，它是这样做的：

```
textproperty::textproperty(LPSTR aName)
 :      customproperty(aName,CString)
 {
 ElementSize=sizeof(text);
 }
```

text 类已经实现了右边接标准 C 风格的字符串的赋值操作符：

```
LRESULT textproperty::SetValue(control & Control,LONG Value,LONG Index)
 {
 tex* Storage=(text* )Control. GetStorageAddress(GetStorageOffset(Index));
```

```
*Storage=(char far*)Value;
return 0;
}
```

不幸的是，C 字符串并不容易处理。但 **text** 类仍定义了向 **const char*** 的类型强制转换，所以我们可以使用 strcpy()函数：

```
LRESULT textproperty::GetValue(control & Control,LONG & Value,LONG Index)const
{
text*Storage=(text*)Control.GetStorageAddress(GetStorageOffset(Index));
strcpy((char far*)Value,*Storage);
return 0;
}
```

LoadValue()实际上是非常有趣的，因为它必须要读入的并不是 text 对象，而是不知长度的字符串，再将这个字符串赋值给 text 对象：

```
LRESULT textproperty::LoadValue(control & Control,HFORMFILE File)
{
UINT Length;
char*AsStored;
LRESULT Error(0);
text*Storage=(text*)Control.GetStorageAddress(GetStorageOffset());


Error=VBReadFormFile(File,&Length,sizeofLength);
if(!Error)
    {
    AsStored=new char[Length];
    Error=VBReadFormFile[File,AsStored,Length);
    }
if(!Error)
    *Storage=AsStored;
return Error;
}
```

正如你所看到的，这个函数首先从盘中读入长度，再读入字符，并且不带 NULL 结尾符。NULL 结尾是后加上的。显然 SaveValue()必须是相反的操作：

```
LRESULT textproperty::SaveValue(control & Control,HFORMFILE File)const
{
UINT Length;
LRESULT Error(0);
text*Storage=(text*)Control.GetStorageAddress(GetStorageOffset());
```

```
Length=(UINT)Storage->GetLength();
Error=VBWriteFormFile(File,&Length,sizeofLength);
if(!Error)
    Error=VBWriteFormFile(File,(void*)(const char*)*Storage, Length);
return Error;
}
```

6.5.5 pictureproperty(图片属性)类

pictureproperty(图片属性)类又是一个有趣的类，因为它与 textproperty 类相反，根本就不处理属性的存贮。Visual Basic 已经处理了 HPIC 格式的内部数据的存贮。所以，并不需要编写一组复杂的函数来处理这种格式，我们所要做的就是在构造函数中控制 PROPINFO 标志：

```
pictureproperty::pictureproperty(LPSTR aName)
  :     customproperty(aName,Picture)
  {
  PropertyInfo.fl=(FLONG)Picture|
      PF_fGetData|PF_fSetData|PF_fSaveData;
  PropertyInfo.offsetData=GetStorageOffset();
  ElementSize=sizeof(HPIC);
  }
```

那些标志通知 Visual Basic(或 VBX 管理器)替你处理数据存贮。这意味着这个类的对象永远也不会接收到 VBM_SETPROPERTY、VBM_GETPROPERTY、VBM_SAVEPROPERTY 和 VBM_LOADPROPERTY 消息。所以我们就不必提供这些消息的处理器了。

6.5.6 floatproperty(浮点数属性)类

这个类之所以有趣的唯一原因就是你必须设法从 lParam 参数中获取或向 lParam 参数设置 float 类型值。问题是由 C 带来的：C 中一般不进行 long 和 float 之间的类型转换，但它们是可以进行转换的。所以我们是用这处方式来骗过 C/C++编译器的：首先将 long 类型强制转换为地址，再由地址获得 float 类型数据：

```
LRESULT floatproperty::SetValue(control & Control,LONG Value,LONG Index)
  {
  float*Storage=(float*)Control.GetStorageAddress(GetStorageOffset(Index));
  *Storage=*((float*)&Value);
  return 0;
  }
```

用相反的过程我们可以获取属性的值：

```
LRESULT floatproperty::GetValue(control & Control,LONG & Value,LONG Index)const
  {
  float*Storage=(float*)Control.GetStorageAddress(GetStorageOffset(Index));
```

```
*((float*)&Value) = *Storage;
return 0;
}
```

6.6 属性列表

在我们结束对属性的讨论之前，还要介绍一个类：propertieslist 类。这是个独立于软件商的容器类，它是一个可以插入属性对象的有序数组。在前面我们见过，model 类使用了这个类。它有一个 propertieslist 属性。实际上，model 就是通过 propertieslist 属性来维护它的属性。

下面是 propertieslist 类的说明：

```
class propertieslist
  {
  public:
      enum {MaxProperties=128};
      propertieslist(USHORT aRunVersion=VB300_VERSION);
      propertieslist & operator+=(standardproperty & Property);
      UINT Find(standardproperty & Property);
      customproperty & operator[](int i);
      NPPROPLIST GetList(void) {return(NPPROPLIST)PropertyList;}
      USHORT GetCount(void) {return Count;}
  private:
      USHORT RunVersion;
      USHORT Count;
      standardproperty*Properties[MaxProperties];
      PPROPINFO PropertyList[MaxProperties];
  };
```

从使用角度来看，propertieslist 类非常类似于 PROPINFO 指针数组；PROPINFO 指针存贮在 MODEL 结构中并且 VBRegisterModel()要用到它。在 C 语言的 VBX 中，这个数组在编译时就已被分配空间了。而 propertieslist 类只有在运行时才可以加入属性。

由于当向列表中加入属性时要动态分配一个内存块，所以为了避免过多分配内存所带来的负担，我规定了一个控件最多只能有128个属性。这也是最恰当的数量限制，因为过多的属性会把用户搞糊涂的。

model 对象在初始化它的属性时，会把 VBX 管理器的当前版本号传递给 propertieslist 的构造函数：

```
propertieslist::propertieslist(USHORT aRunVersion)
  : RunVersion(aRunVersion), Count(0)
  {
  PropertyList[0]=NULL,
  }
```

+=操作符可以用来向列表加入属性。注意，只有当属性的版本低于或等于当前 VBX 管

理器的版本时，这个属性才可以被加入：

```
propertieslist & propertieslist::operator+=(standardproperty & Property)
  {
  if(Property Version<=RunVersion &&(Count<MaxProperties))
    {
    Properties[Count]=&Property;
    propertyList[Count]=Property. PropertyID;
    PropertyList[++Count]=NULL;
    }
  return *this;
  }
```

这可以避免版本不匹配的问题。当试图在只支持 VBX1.0的 Microsoft Visual C++中使用 VBX3.0时就会遇到版本不匹配的问题。

还要注意的是，我们在将属性对象加入 Properties 数组的同时，还将一个指针加入 PropertyList 数组中。PropertyList 最终要通过 VBRegisterModel()函数被利用起来。它必须是一个连续的指针数组。

有时需要向 propertieslist 对象查询给定属性在列表中的位置。这个任务是由 Find()方法来完成的：

```
UINT propertieslist::Find(standardproperty & Property)
  {
  for(register UINT p=0;p<Count;p++)
      if(& Property==Properties[p])
          return p;
  return Count;
  }
```

下标操作符返回相应属性的引用：

```
customproperty & propertieslist::operator[](inti)
  {
  return(customproperty &) * (Properties[i]);
  }
```

这样，如果控件是由 model 派生类表示的，我们就可以通过 model 类来很好地控制与这个控件相关联的属性了。

6.7 特殊事件和一般事件

将属性类和事件类放在同一部分介绍的原因是它们在很多方面很相似。事件比属性要简单一些，但属性是通过消息来设置的，所以属性包含在 model 类和 control 类中；而事件必须与系统保持联系，因此只包含在 model 类中。

触发一个事件需要 control 类的对象，并且只有这样才能知道是哪个控件触发的事件。

与属性类似，事件都是基于一个标准类的，而用户事件则要基于一个派生类。这个标准类

是 standardevent 类：

```
class standardevent
  {
  public:
      standardevent(PEVENTINFO aStdEvent,
            USHORT aVersion=VB100_VERSION);
      virtual BOOL IsStandard(void)const;
      ERR Fire(control & Control,LPVOID Params);
  private:
      PEVENTINFO EventID;
      USHORT Version;
  friend class eventslist;
  }
```

和 standardproperty 类一样，standardevent 类构造函数只是保存事件 ID 和版本，使这个类可以适于所有标准事件：

```
standardevent::standardevent(PEVENTINFO aStdEvent,
          USHORT aVersion)
    : EventID(aStdEvent), Version(aVersion)
    {
    }
```

同样也象 standardproperty 对象那样，standardevent 对象规定自己是标准的，虚函数 IsStandard()会返回真：

```
BOOL standardevent::IsStandard(void)const
  {
  return TRUE;
  }
```

Fire()方法只是封装 VBFireEvent()函数：

```
ERR standardevent::Fire(control & Control,LPVOID Params)
  {
  UINT EventID=Control.GetModel().Events.Find(*this);
  return VBFireEvent(Control,EventID,Params);
  }
```

虽然标准事件很少被显式地触发(fire)，系统会替你触发它们；但 Fire()被设计成 standardevent 类的一个方法，做为对诸如鼠标按键和键盘击键等预定义事件的响应。但如果你真的需要的话，你可以亲自来触发这些事件。

用户事件的封装要有趣得多，因为我花了一些时间在 customevent 类中建造了一些工具函数，这使它比用 C 实现的更易使用。下面是它的说明：

```
class customevent: public standardevent
  {
  public:
```

```
        customevent
            (
            LPSTR aName,
            LPSTR aParamList,
            BOOL UnloadOk=TRUE
            );
        virtual BOOL IsStandard(void)const;
    private:
        char Name[32];
        char ParamList[256];
        void ParseParamList(void);
        EVENTINFO EventInfo;
        WORD ParamTypes[12];
    };
```

就象 customproperty 类封装 PROPINFO 结构那样，customevent 类封装了 EVENTINFO 结构。所以，它的构造函数需要初始化这个结构：

```
customevent::customevent
        (
        LPSTR aName,
        LPSTR aParamList
        BOOL UnloadOk
        )
: standardevent((PEVENTINFO)&EventInfo, 0)
{
stropy(Name,aName);
EventInfo. npszName=(PSTR)Name;
EventInfo. npParmTypes=(PWORD)&. ParamTypes;
strcpy(ParamList,aParamList);
EventInfo. npszParmProf=(PSTR)ParamList;
ParseParamList();
EventInfo. fl=UnloadOk?0: EF_fNoUnload;
}
```

你会看到，在构造函数的参数列表中有一个字符串指针 aParamList。这个指针在存入 EventInfo 结构的同时，另一个域 EventInfo. npParmTypes 也被初始化了。那个地址从何而来？更确切地说，它是怎么获得有效值的呢？

严格地讲，在赋值时，它并不是一个有效的值。但在它被 VBX 管理器引用之前会变成有效值，这要归功于成员函数 ParseParamList()：

```
void customevent::ParseParamList(void)
    {
    EventInfo. cParms=0
```

```
register ParamListLength=strlen(EventInfo.npszParmProf);
for(register i=5;i<ParamListLength;i++)
    if(toupper(EventInfo.npszParmProf[i-4]) =='"&&
            toupper(EventInfo.npszParmProf[i-3]=='A'&&
            toupper(EventInfo.npszParmProf[i-2]=='S'&&)
            toupper(EventInfo.npszParmProf[i-1]=='')
        switch(toupper(EventInfo.npszParmProf[i]))
            {
            case 'I':
                ParamTypes[EventInfo.cParms++]=ET_12;
                break;
            case 'L':
                ParamTypes[EventInfo.cParms++]=ET_14;
                break;
            case 'S':
                if(toupper(EventInfo.npszParmProf[i+1]) ==T)
                    ParamTypes[EventInfo.cParms++]-ET_R4;
                else
                    ParamTypes[EventInfo.cParms++]=ET_HLSTR;
                break;
            case'D':
                ParamTypes[EventInfo.cParms++]=ET_R8;
                break;
            case'C':
                ParararnTypes[EventInfo.cParms++]=ET_CY;
                break;
            }
EventInfo.cwParms=EventInfo.cParms*2;
}
```

你会记得EVENTINFO结构中包含有两个并行的数据：一个参数(字符串)列表和那些参数的数据类型的列表(整型数组)。程序员完全有可能漫不经心地把这两个列表弄得相互不一致。而ParseParamList()函数的任务就是读入参数描述字符串并从中得出参数的数据类型再放到数据类型列表中。

evenslist类与propertieslist类几乎是一样的：

```
class eventslist
  {
  public:
      enum {MaxEvents=32};
      eventslist(USHORT aRumVersion=VB300_VERSION);
      eventslist &operator+=(standardevent & Event);
```

```
        UINT Find(standardevent & Event);
        standardevent & operator[](int i);
        NPEVENTLIST GetList(void)
            {return(NPEVENTLIST)EventList;}
    private:
        USHORT RunVersion;
        USHORT Count;
        standardevent* Events[MaxEvents];
        PEVENTINFO EventList[MaxEvents];
    };
```

我们规定最大的事件数32。构造函数只是做了准备对象的工作，这里准备了一个空容器：

```
eventslist::eventslist(USHORT aRunVersion)
    : RunVersion(aRunVersion), Count(0)
    {
    EventList[0]=NULL;
    }
```

并且+=操作符可以用来向容器中加入 event 对象：

```
eventslist & eventslist::operator+-(standardevent & anEvent)
   {
   if(anEvent. Version<=RunVersion &&
            (Count<MaxEvents))
       {
       Enents[Count]=&anEvent;
       EventList[Count]=Events[Count]->EventID;
       EventList[++Count]=NULL;
       }
   return*this;
   }
```

Find()和下标操作符同样是与 propertieslist 类中相应函数相似的。

6.8 去做吧

我是用一个简化了的使用 VBX++来创建叫做 Gradient 的控件的例子作为本章的开始的。为了演示我们已开发类的使用，让我们来用这个名字创建一个真正的控件吧。

gradient 控件试图用来加强控件的外观显示。它把自己画成逐渐变暗的一个象素宽的一组线，这样控件的顶部是用户所选择颜色的完全显示，而底部则是全黑，中间是平缓的过渡。

当然，第一步是为新控件创建一个工程。加入资源文件、模块定义(DEF)文件、一个 CPP 文件(比如,GRADIENT. CPP)等。此外,这个工程中还需要加入两个库：一个当然是 VB\CDK\VBAPL. LIB；另一个是 VBXXX. LIB，它包含前面三章我们所开发的类。

模块定义文件类似于下面几行：

```
LIBRARY       GRADIENT
DESCRIPTION   'Gradient VBX Custom Control'
EXETYPE       WINDOWS
CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE SINGLE
HEAPSIZE      4096
```

以上工作做好后，我们可以回到GRADIENT.CPP模块。它以一个#include语句和gradient类的说明开头：

```
#inolude"vbxxx.h"

chass gradient: public model
  {
  public:
      gradient();
  protected:
      virtual FLONG GetModelFlags(void);
      virtual void OnDraw(control & Control,devicecontext & Dc,
          BOOL Erase,rectangle Rect);
  private:
      colorproperty TopColorProp;
  };
```

为了使#include<vbxxx.h>有效，别忘了把VBXXX目录（文件夹）加入到包含文件和库文件的目录查询列表中。

gradient类中定义了三个方法和一个属性。这些就足够了。我们设法使说明一个新控件变得容易些。那么是如何实现的呢？

第一个任务是构造那个控件：

```
gradient::gradient()
  :    model((PSTR)"Gradient"),
       TopColorProp((PSTR)"TopColor", RGB(255,255,255))
  {
  Propcrties+=*Top;
  Properties+=*Left;
  Properties+=*Height;
  Properties+=*Width;
  Properties+-TopColorProp;
  }
```

控件的名字被传递给基类的构造函数，用户属性也被构造了；并且所有的这个控件要支持的属性，包括标准的和用户的，都被加入到继承的Properties集合中。（这个控件并不触发事件，如果需要这个控件触发事件，只需将事件加入Events集合中就可以了。）

因为为了减小负担，我们把 gradient 设计成图形控件，所以 GetModelFlags()函数被重载了：

```
FLONG gradient::GetModelFlags(void)
  {
  return MODEL_fGraphical;
  }
```

OnDraw()函数与其说是 VBX++的演示，不如说是 GDI 类的演示。它的任务就是从控件的顶部向底部画线，每一条线都比它上面的线暗一些，直到最底部的线完全变成黑色：

```
void gradient::OnDraw(control & Control,
        devicecontext & Dc,BOOL,rectangle Rect)
  {
  color TopColor((COLORREF)TopColorProp.GetValue(Control));
  color LineColor;
  register Percent;
  register Offset(Rect.top);
  Rect.top=0;
  Rect.bottom-=Offset;

  for(registeri=0;i<Rect.bottom;i++)
    {
    Percent=100-((i*100)/Rect.bottom);
    LineColor=
      red((BYTE)((red)TopColor)*(BYTE)Percent/100);
    LineColor=
      green((BYTE)((green)TopColor)*(BYTE)Percent/100);
    LineColor=
      blue((BYTE)((blue)TopColor)*(BYTE)Percent/100);

    pen Pen(LineColor,2,pen::InsideFrame);
    Dc=Pen;
    Dc.MoveTo(point(Rect.left,Offset+i));
    Dc.LineTo(point(Rect.right,Offset+i));
    }
  }
```

当你意识到图形控件要使用父窗口用户区域坐标时，你就会明白为什么要使用局部变量 Offset 了。这样 Rect.top 不可能是0。为了计算方便，这里改变了 Rect 的值。

接下来我们来画每条线。每个基色(红、绿、蓝)都是从 TopColorProp 属性值和当前线距离顶部的百分比计算得出的。一旦我们得到了那个颜色，我们就可以构造一个那种颜色的画笔对象了。在生成画笔时，如果设置了 pen::InsideFrame 风格并且笔的宽度大于1，那么这个笔就将是一个抖动的笔。LineTo()的效率要比 DrawRectangle()高，所以这里使用了 LineTo()函数。

画笔是通过使用赋值操作符被选入 devicecontext 对象的，接下来就是画线。这时代码已经到了 for 语句的结尾处，因此 pen 对象结束了它的生命周期。在画笔对象被撤消之前，如果画笔被选入设备环境中，画笔的析构函数会自动从设备环境中删除它。

最后，我们需要通过生成 gradient 类的一个实例来促使以上这些操作的发生：

gradient Gradient；

这行语句给了我们所要的一切：GRADIENT. VBX 诞生了！我们可以把它加入到一个工程的 Visual Basic 控件列表中；我们可以把它放在一个表中；我们不但可以设置 TopColor 的值，也可以设置它在表中的位置。图6.4就是一个例子。

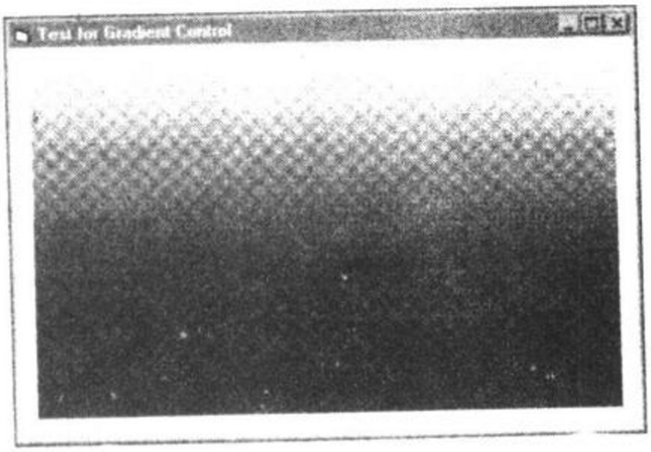

图6.4　Gradient 控件使用的例子

第三部分 VBX 生成器

第七章 介绍 VBX Genie

在 VISUAL C++和 Borland C++之前，我对 CASE(计算机辅助软件工程)工具从没有什么好感。其中一个原因是我并不相信“软件工程”这一套，我更相信软件艺术。作为软件艺术家，我们当然应该懂得怎样使用我们的工具，正象是一位画家需要了解他或她作画的颜料、画笔和颜料溶剂的性能。但是，(在 CASE 中)实际的组装工作并不仅仅是组装。CASE 最初许诺要强化“简单”编程规则，这样即使是普通程序员也可以“揣摩”出高水平的应用程序。而结果是，那些工具确实能够揣摩，但却并不能帮助所有的人生成高水平的应用程序。普通程序员仍然只能写出一般化的应用程序，而艺术家级的程序员则被诸如“一个应用程序应该如何布局”这样的平庸学院式见解弄得心烦意乱、哭笑不得。

Visual C++ 和 Borland C++IDE(集成开发环境)是称职的 CASE 工具，但它们走的是另外一条道路：它们根据您的指点生成框架代码，然后就主动让开道路。它们是一些工具的集合，而不假装为自命不凡的程序员高手—我佩服高手，但我要尽可能地赶超他们。例如，我们知道，象 VBX 这样的软件部件，要建构起它的基本结构并不复杂；但这确实是一件很烦琐的工作。它有一个 MODEL 结构、有属性和事件列表的构造，还有那么多不同的消息处理函数的头文件。既然计算机能够根据您提出的要求替您生成这些代码，那为什么还非得要劳您把它们敲进去，或者剪切和粘贴呢?

编写一个功能齐全的 VBX 生成器并不容易。但能够拥有一个生成器还是不坏的，况且学会怎样编写代码生成器也很值得!所以在这一部分的以下各章，我们要学习代码生成器 VBX Genie 的设计与实现的各个方面。学完之后，您将不但拥有一个 VBX 代码生成器，您还会懂得如何编写自己的代码生成器，而且还积累了一些可重用的工具，可为您编写自己的代码生成器助一臂之力。

7.1 代码生成器的设计

一个代码生成器包括以下三个基本部件：

●代码模板。这是您想要生成的代码，同时也包含有某种方法把某些部分替换为生成的代码。

●生成代码的模块，该模块生成的代码与代码模板的其余部分融合。

●获取信息的模块，所获取的信息在实际生成代码时要用到。

有时，在一个非常简单的代码生成器中，上面的一个或多个部件可能会略去。例如，如果您只想生成 C++类的头文件这样的小代码片断，就用不着代码模板。或者，如果可替换信息

不需要用户输入(例如可以从系统获得的时间和日期)就可能省去用户界面。但在本书的 VBX Geniez 中，所有这三个部分都要用上。在编写 Genie 的代码之前，我们先来更细致地讨论一下这三个部件。

7.1.1 代码模板

VBX Genie 所要生成的基本代码正如我们见到过的：根据用户的要求，生成 VBX 框架和(或)VBX++类库。但我们必须允许对生成后的代码作程序性的修改。修改的对象包括属性和事件的定义，以及处理 VBM_METHOD 消息的可选函数。对 VBX 框架来说，修改必须在代码之中完成；对 VBX++来说，除了类库部件之外，还应由生成器产生新控件派生类的文件。

所有的代码修改(至少在这里!)都可分为两类。第一类，简单的文本替换。这类似于全程查找和替换操作。例如，您可以把一套文件作为某一类型工程的基础。在那些文件里，可能会引用到工程名称，然而工程名称是随不同的工程而各不相同的。所以在最初的文件中，在工程名称出现的地方可用字符串“$PROJECTNAME”代替。然后当做完拷贝之后，您可以使用 IDE 的查找和替换机制把出现“ $PROJECTNAME”的地方改为工程的正确名称。

这正是我们在 VBX Genie 中所采用的方法(当然我们要使这个过程自动化)。

第二类修改是包含(或剔除)可选代码块。这些代码块只是在某些环境的最终文件中才出现。在 C 和 C++中做到这点并不难，可用 #define、#ifdef、#ifndef 宏。我们将用某种类似的机制实现。

顺便说一句，在我编写代码生成器的过程中，我遇到过另一类的模板修改，即循环代码。这相当于是在模板中包含了“循环”；“循环”内的代码根据代码生成器所提供的变量值被复制，同时循环索引变量的当前值被插入到生成代码内的适当位置。但是由 VBX Genie 所生成的代码用不着这种修改。

7.1.2 代码生成器

对模板生成的代码拥有控制修改权是很重要的，在对此有所了解之后，您就需要有一个代码生成器来实际生成代码并完成修改工作。如果代码模板存贮在外部文件中，就必须要先找到这个文件并把它读入内存(如果找不到这个文件，还应处理这一错误)。一个好的办法是把模板文件作为程序员定义的资源附加到 EXE 文件中，这样就不会丢失了。不管用哪种方法，只要已读入内存，模板“变量”(我把它们叫做替身)就必须要被替换成适当的值；同时，剔除掉那些没有被选中的可选代码。然后文件就生成了。

由于这项工作所带有的普遍性，因而这里实现代码生成器的模块可以用于任何代码生成器的设计，而不仅仅局限于 VBX Genie。

7.1.3 用户界面

“Genie”是一类“工具”性质的应用程序。这样的应用程序既不经常运行，而且也只是短期使用(与字处理软件相比)，因此它的界面必须要适合于偶尔使用的要求。也即是说，我们不能指望用户成为 VBX Genie 专家!在设计界面时，应该把注意力放在条理的清晰程度，而不是访问速度上。

在 Windows 中，设计工具类应用程序常使用两种方法。一种是使用一系列的面板或对话框，其中包含“Next”(下一个)和“Previous”(上一个)按钮。通常还有“Finish”(完成)和

"Exit"(退出)按钮。这种设计方式最适用于固定的、按部就班的工作。如果需要，用户可以返回上一步(尽管这样做令人不快)。

第二种方法允许用户同时对所有部分进行访问。Windows 95用户界面在其"属性页"的概念中用到这一技术。它的想法是，需要完成的工作其实只有一步；然而需要的选项太多，在一个对话框中安置不下。选项之间没有先后之分，所以不好用一系列面板来实现。在属性框中，把要修改或创建的方面和属性按逻辑分组，并且排列成标以标签的子对话框，这些子对话框被称为页。

这第二种方法完全符合创建 VBX 的要求，所以我们设计的 VBX Genie 就采用这种方法。图7.1是 VBX Genie 的外观，显示的是 Project(工程)属性页。

图7.1 VBX Genie 外观

与其他 Windows 95 对话框一样，VBX Genie 看起来简洁明快，可以毫不费力、毫不混乱地访问各个组属性。作为工具类应用程序，它的重点并不是放在处理文档上。它创建文件，但不能编辑文档。因此，VBX Genie 没有 File(文件)菜单。实际情况是，没有任何菜单。尽管 VBX Genie 是一个 EXE 文件，但是在用户看来，它只是对话框。

用户选定所需选项之后，再单击 OK 按钮，工作就正式开始了。VBX Genie 寻找或者（如果没有，就）创建指定的目录（在 Windows 95中叫做"文件夹"），在这个目录中创建几个文件。VBX Genie 作为一种工具，最好是加入到编程环境的工具菜单中，它将和 IDE 打交道，为这个新的 VBX 创建相应的工程并打开该工程。

7.2 选择一门实现 VBX Genie 的语言

VBX Genie 本来可以用 MFC 或 ObjectWindows 编写。然而，尽管这两种语言工具都可以毫不含糊地完成这一任务，但 VBX Genie 工程关心的重点是用户界面。而且 VBX Genie 还应该控制 Visual C++ 或 Borland C++，指挥它创建出新的工程。由于这些原因，我们说，在本工程中，这两种语言工具的 C++类库都不能充分施展其才能。而 Visual Basic 则可以在注重用户界面的场合大显身手。而且 Visual Basic 40已在近期发行，它所具有的新特色使得我们这个工程实现起来更加容易。

VB4.0除了能够把资源加入到 EXE 文件中外，还支持程序员定义的对象，这一点类似于

C++。只能说是类似于C++，因为它不支持继承。幸而C++85%的价值在于简单的封装，而VB中的对象可以很好地实现封装。

图7.2显示的是我们将要用到的顶层对象。

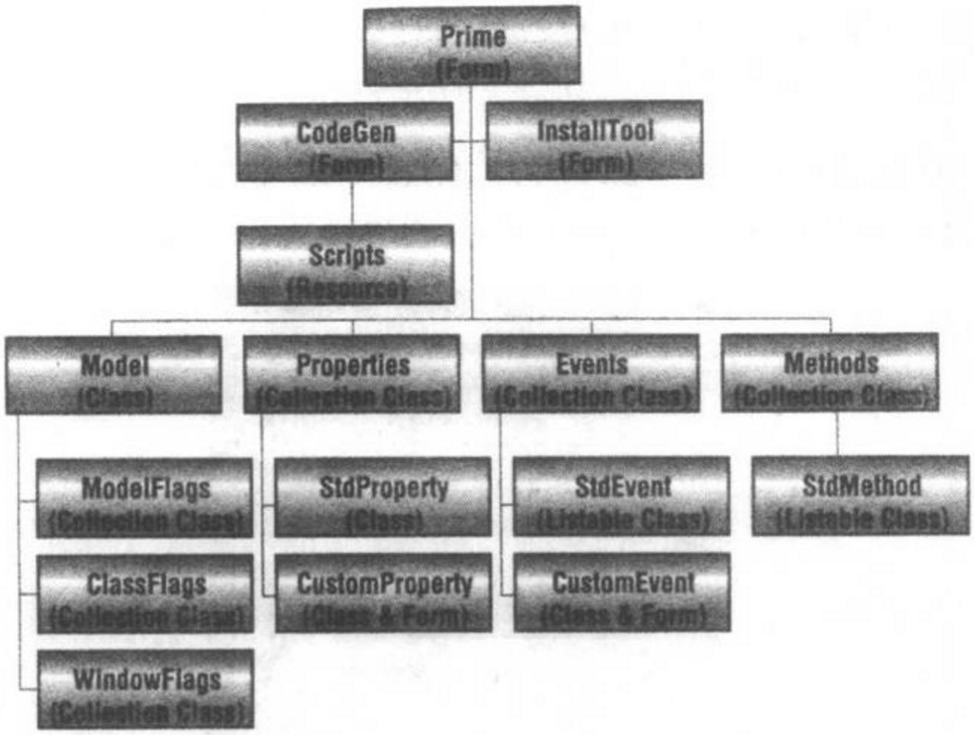

图7.2　实现 VBX Genie 要用到的顶层对象

注意，图7.2给出的是功能性布局图，它展示出各主要对象，但并未显示出它们之间的继承性层次关系。下面是重要的注意事项：

●主表 Prime 控制所有的一切。

●代码生成器在 Prime 中调用，它做的大部分工作就是让各个对象编写自己的代码。也就是说，Properties(属性)类编写属性代码，Event(事件)类编写事件代码等等。

●代码生成器把它的脚本(即代码模板)保存在应用程序的 resource pool(资源池)中，VB4的新特色可实现这一点。

●Listable 类被六个不同的类所使用。这是活生生的代码重用的例子！

●对象图与用户界面图直接对应。表与函数相随。

在下面的几章中，我们要逐行实现 VBX Genie。在此过程中，请把图7.2放在方便的地方。VBX Genie 是个不寻常的程序。但是我坚信，如果采用了适当的面向对象的方法，那么即使是最复杂的应用程序也可以有简洁的外表，而且最终也可以象简单程序一样具有鲁棒性(robust)。不信我们走着瞧！

第八章　创建用户界面

我们已经为 VBX Genie 的开发作了铺垫，下面可以开始设计和编写代码了。有许多事情要做；在本章和下面的两章中，我们要开发出所有的代码。

在 VBX Genie 运行的时候，我们最先看到的是用户界面，因此我们首先设计用户界面并编写出它的代码。把编写用户界面代码的工作放在前面还有一个原因，即它是最容易建立的部件。由于 VBX Genie 的特点是高度可视化和交互性，所以我们要另换一种工具 Visual Basic，不再使用 C＋＋。这带给我们的好处是：我们将创建出功能强大的工具，它的界面可以轻松地修改，而且还可以扩充进其他的特性。

我们从主表以及 VBX 赖以工作的程序员定义类开始讨论。然后为界面的每个标签页编写代码。这样安排的好处是，我们建立界面的过程与用户创建 VBX 软件部件时与代码生成器交互的过程是一致的。

8.1　建立主表 Prime

回忆第七章的内容，VBX Genie 被设计成一个对话框，它包括六个可选标签：

- ●Project(项目)　　●Version(版本)
- ●Model(模型)　　●Properties(属性)
- ●Events(事件)　　●Methods(方法)

用户可以选择以上的各个标签来设置 VBX Genie 的不同选项。要建立这样一个界面，我们先从主表 Prime 入手。

在使用 Visual Basic 的早期，我就开始把我的启动表命名为“Prime”。其实在最初的时候，我是把它叫做“Main”，但是我发现如果存在着一个名字为“Main”的子过程，那么“Main”表和“Main”子过程会相互影响，所以不得不忍痛割爱，另换一个名字。而象“Forml”之类的命名方法对我来说则从来不在考虑之列！

Prime 有一个对话框边框，而且因为它并不在屏幕上停留很长时间，故而没有菜单。Micrososft 公司把这样的程序叫做“wizards”(大师)，Borland 公司把它们叫做“experts”(专家)，我则把它们叫做“genies”(精灵)。设计这些程序的目的就是要完成专门而集中的工作，完事之后它们就退居二线。VBX Genie 将以工具的形式从 Microsoft Visual C＋＋1. 5x(16位版本)或 BorlandC||4. 52(这是个能够生成无错机器代码的版本)的工具菜单上运行。

VBX 必须是16位的动态链接库，没有必要把 VBX Genie 编写成能够运行在32位 Windows 下。因此，VBX Genie 是在 Visual Basic4. 0的16位版本下完成的(专业版带有该版本)，并且在构造 VBX Genie 时可以使用 VBX 控件(比如 InitData 控件，它是在我的另外一本书 Windows Programming Power with Custom Controls 中实现的)。

坦率地讲，需要搜集的信息太多了。如果全都堆放到一个面板里则会显得太挤。有两套解决方案可供选择，一是多页式(multi_page_style)，Microsoft 公司在许多 Wizard 程序中采用这种方案；一是标签式(tabbed style)，在属性页中和许多别的对话框中采用。我选用的是后一种，目的是让用户能够方便地在各种选项之间前后跳转。

8.1.1 编写 Prime 表的代码

在第七章里给出的功能图中我们可以看到，Prime 表需要管理(或至少是拥有)好几种类型的对象。这些对象分别表示了目标 VBX 的模块标志(model flags)、类风格标志(class flags)、窗口风格标志(window class flags)、属性、事件、方法。它们被说明在 Prime 的(General)(Declarations)部分：

```
Option Explicit
Dim ModelFlags As New ModelFlagsClass
Dim ClassFlags As New ClassFlagsClass
Dim WindowFlags As New WindowFlagsClass
Dim Properties As New PropertiesClass
Dim Events As New EventsClass
Dim Methods As New MethodsClass
```

各条语句中的关键字 New 意味着，在 Prime 表初始化的时候，这些对象也将被初始化。与 C++的对象类似，这里的对象也负责自身的构造工作，这样一旦 Prime 准备执行代码，各对象也准备就绪。

在装入表之前，在它被初始化的时候，有一个新的事件 Initialize(初始化)将被触发。Initialize 的存在使得表与类对象有了更多的共同点。(类对象也有 Initialize 事件，当然，它没有 Load 事件。)然而，对表来说，Initialize 的作用是有限的，因为此时表还没有装入，控件还不存在。因为这个原因，Prime 在 Load 事件中才出现它最初的可执行代码：

```
Pirvate Sub Form Load()
IniDatal. filename=Makepath("",App. Path,App. EXEName,". ini")
If Not CheckInstallation Then
        Unload Me
        Exit Sub
End If
Dim c As WaitCursor
Set c=New WaitCursor
LoadVersionDefaults
ModelFlags. SetListbox lst_ModelFlags
ModelFlags. SetProperties Properties
ClassFlags. SetListbox lst_ClassFlags
WindowFlags. SetListbox lst_WindowFlags
Properties. SetListbox lst_Properties
Properties. SetModelFlags ModelFlags
Properties. SetEvents Events
Events. SetListbox lst_Events
Events. SetProperties Properties
Events. SetMethods Methods
Methods. SetListbox lst_Methods
```

```
Methods. SetEvents Events
cmb_VBVersion. ListIndex=2
End Sub
```

第一条语句是把 VBX Genie 的. INI 文件的完整路径提供给 IniData 控件。这样，VBX-GEN. INI 就可以不必放到杂乱的 WINDOWS 目录下。

函数 AlreadyInstalled 检查 VBX Genie 是否已经被安装到集成环境的工具菜单当中。如果还没有安装，它会假设您现在运行这个程序的目的是为了安装它，于是就安装自己，然后返回 False。否则就返回 True，接着做下面的工作。

在等待显示表的时候，VB 不能自动地把光标替换为沙漏光标，有 WaitCursor 类的那两行代码就服务于此目的。WaitCursor 类是一个可重用部件，在它被创建的时候会把光标变为沙漏，而在作用域之外则把光标复原。

由于某种古怪的原因，如果 WaitCursor 类只是用 Dim 作简单的声明，或者甚至用上 New 而不实际地引用，它都不能正常工作。显然这是某种优化机制在作怪；VB 的项目师们宁可把可贵的时间花在“优化”上面，也不好好反省一下“短路”的表达式，真是糟糕。对不起我扯远了。

用默认值设置好 Version 页之后，下面就是把(General)(Declaritions)中声明的各个对象与各自的列表框相联系，并且建立起对象之间的联系以支持 MODEL 标志、属性、事件和方法之间的交互作用。(当然，在为这些对象编写代码的时候，我们会对这些函数作更加细致的研究。而现在，您只需要认定这些函数能够完成其函数名称所表明的那些功能即可。)

最后一行代码选定组合框的第二个索引项。该组合框位于 Model 页，列出了现已发售的 1.0、2.0和3.0三种版本的 VBX 管理器(VBX manager)。并不需要有专门的代码把这三个值放入组合框，因为在设计时就可以做好。

我们其实并没有在某个地方装入那些列表框。那么是这些神奇的类在做装入工作吗?不是。是 version 组合框触发了装入过程。因为只要用户选择另一个版本的 VBX 管理器，列表框就得重新装入。当有改动时，不论这种改动是由用户操作引起还是由程序引起，结果都会触发组合框的Click(单击)事件，因此我们可以利用这一点来装入，或者重新装入所需的列表框：

```
Private Sub cmb_VBVersion_Click()
ModelFlags. TargetVersion=Val(cmb_VBVersion)
ModelFlags. LoadListbox
ClassFlags. TargetVersion=Val(cmb_VBVersion)
ClassFlags. LoadListbox
WindowFlags. TargetVersion_Val(cmb_VBVersion)
WindowFlags. LoadListbox
Properties. TargetVersion=Val(cmb_VBVersion)
Properties. LoadListbox
Events. TargetVersion=Val(cmb_VBVersion)
Events. LoadListbox
Methods. TargetVersion=Val(cmb_VBVersion)
Methods. LoadListbox
```

```
End Sub
```

这些用户自定义类每个都有一个 TargetVersion 属性，它将(显然地!)决定在列表框内放入哪些项。

8.1.2 标签控件(Tab Control)的一些问题

在逐页描述本应用程序之前，还有一个与标签控件有关的问题需要讨论。这就是 SSTab 控件，由 SheridanSoftware 软件公司编写，并与 Visual Basic 4.0捆绑发售。SSTab 相当出色，可以说是我所用过的控件中最好的一个。(它肯定要超过 Microsoft 的"属性页"标签控件，因为属性页仅仅是标签控件而已，它不是控件容器，没有多大用处。而且属性页控件还不能运行于16位的应用程序中。)

当用户从一页跳转到另一页时，SSTab 控件将经历 GotFocus 事件，即使控件本来拥有输入焦点也是这样。所以在 GotFocus 事件处理函数中有一小段代码用来把焦点重新设置到刚激活页的第一个控件上。

```
Private Sub SSTabl_GotFocus()
On Error Resume Next
Select Case SSTabl. Tab
Case 0:
  txt_Name setFocus
Case 1:
  txt_FileVersion(0). SetFocus
Case 2:
cmb_VBVersion. SetFocus
Case 3:
lst_Properties. SetFocus
Case 4:
lst_Events. SetFocus
Case 5:
lst_Methods. SetFocus
End Select
End Sub
```

在开头使用 On Error 语句的原因是，当表首次变为活动状态时，它所包含的控件不可能有焦点切换，而此时仍将激活 GotFocus 事件。这个错误没有什么危害性，可以安全地忽略掉。

此外还有一个与标签页有关的问题是需要处理表的 KeyDown 事件。SSTab 控件用 Control|Tab 或 Shift－Control|Tab 翻页，如同在 MDI(多文档界面)应用程序中那样。然而，当 SSTab 上有一些项(否则的话它还有什么用处呢?)，由于当前活动项接收了所有的击键，控件将得不到任何击键。借助 Visual Basic 4.0的另外一个新特色可以解决这个问题：如果您提出要求(通过把表的 KeyPreview 属性设置为 True)，那么对所有的击键事件，表都首先得到通知：

```
Private Sub Form_KeyDown(KeyCode As Integer、Shift As Integer)
If(Shift And vbCtrlMask)=vbCtrlMask Then
If KeyCode=Asc(vbTab)Then
```

```
        Activate Tab SSTabl, ShiftAnd vbShiftMask
        KeyCode=0
    End If
    End If
    If KeyCode=vbKeyFl Then
      About Show vbModal
      KeyCode=0
    End If
    End Sub
```

为了防止击键被再次处理，把 KeyCode 参数设为0。

这里我拦截 F1键，并用它来显示一个 About 对话框。

8.2 支持 Project 页

请回忆第一个标签页“Project”。您也许已经注意到在我们的对象图表中并没有“project”。这是因为 Project 需要存储的信息量极少，可以直接保存在 Prime 表中。(别忘了,VB4中的表是对象,等同于用户自定义类。)

在本页上，不同控件间的交互主要来自第一个信息域，即“Name”(名称)。当用户往该域中键入内容时，内容也同时被复制到本页的“Directory”(目录)域，以及 Version 页和 Model 页上的几个域中。在这里，我们希望仅当别的信息域没有被用户明确地改动时才进行复制。这利用 Tag 属性可实现，几乎所有 VB 控件都具有 Tag 属性。如果它为空，那么对 Name 信息域(这个控件的名称实际上是 txt_Name)的改动就反映到其他控件。而如果用户对这些控件有改动，改动后的值也将被写入 Tag 属性，这样 Name 就知道不要改变它：

```
Private Sub txt Name Change()
Dim aDrive As String
Dim aDir As String
Dim aRoot As String
Dim aExt As String
If txt_Directory. Tag= “” Then
    SplitPath txt_Directory. Text, aDrive, aDir, aRoot, aExt
    txt Directory. Text=LCase $ (MakePath(aDirve,aDir,txt_Name,“”))
End If
If txt_ProductName. Tag= “” Then
    txt_ProductName. Text=txt_Name. Text
End If
If txt_ClassName. Tag= “” Then
    txt ClassName. Text=txt_Name. Text
End If
cmd_OK. Enabled=VerifyOK
End Sub
```

(SplitPath 函数和 MakePath 函数的工作机理类似于 C 标准库中与它们同名的两个函数；函数代码在 FILES. BAS 模块中。)

函数 VerifyOK 检查页上那些可能含有非法值的信息域，并保证只要域中有非法值，OK 按钮就不起作用。只有三个域可能会发生这种情况：在它们各自的 Change 事件中，都调用 VerifyOK：

```
Private Function VernifyOK()As Boolean
VerifyOK=True
If Len(txt_Name. Text)=0 Then
        VerifyOK=False
        Exit Function
End If
If Len(txt ClassName. Text)=0 Then
        VerifyOK=False
        Exit Function
End If
If Len(txt DefControlName. Text)=0 Then
        VerifyOK=False
        Exit Function
End If
End Function
```

VerifyOK 是一个函数。那么它在调用时为什么不加上括号呢?原因很简单一因为 Visual Basic 最终可以做到不需要这些令非 C 语言的程序员头痛的条条框框。这种增强性能是向后兼容的；如果认为有此必要，您尽可以把括号加上。

8.3 支持 Version 页

好，翻过一页，还有五页。Version 页的外观如图8.2所示的对话框，它接收的信息将用于

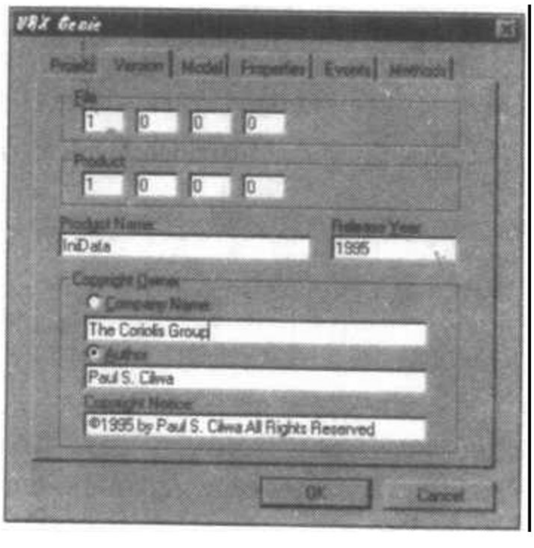

图8.1 VBX Genie 的 Version 页

生成控件的 VERSIONINFO 资源。

"Year"(年份)、"Company Name"(公司名称)、"Author"(作者)和"Copyright Notice"(版权告示)四个信息域之间存在交互作用。属性 txt_Copyright. Tag 的初始值是"@YYYY by OWNER All Right Reserved"。每当 txt_Year、txt_CompanyName、txt_Author 有改动，都将导致 SetCopyright 子过程的调用：

```
Private Sub SetCopyright()
Dim Notice As String
Notice=txt_Copyright. Tag
Dim i As Integer
i=InStr(Notice,"YYYY")
Ifi>0 Then
    Notice=Left $ (Notice,i-1)+txt_Release Year. Text+Mid $ (Notice,i+4)
End If
i=InStr(Notice,"OWNER")
Ifi>0 Then
    Dim Owner As String
    If opt_CopyrightOwnerlsCompany Then
    Owner=txt_CompanyName. Text
  Else
      Owner-txt_Author. Text
  End If
  Notice=Left $ (Notice,i-1)+Owner+Mid $ (Notice,i+5)
End If
txt_Copyright. Text=Notice
End Sub
```

这并不妨碍用户手工地把所感兴趣的内容添加到版板信息域中：

```
Private Sub txt_Copyright_Change()
If txt Copyright Is ActiveControl Then
     txt_Copyright. Tag=txt_Copyright. Text
End If
End Sub
```

实际上，如果用户输入的内容包含有特殊字符串"YYYY"或"OWNER"，那么用户在其他域所作的修改也将对它们作自动更新!

此外还要注意，版板所有者可以是作者，也可以是公司。这两个域都保存在 VERSIONINFO 资源当中，但是只有"owner"(所有者)才会自动出现在版权告示中。

如果每次创建一个 VBX 都需要重新输入公司名和作者名，这显然是个烦恼。VBX Genie 在表面上安放了一个 IniData 控件，当然在运行时您看不到它。该控件(在我写的书《WindowsProgramming Power with Custom Controls》中讨论过，并在本书附带的 CD-ROM 中包括有)可以按 VB 的方式轻松地存取. INI 文件。Prime 表下载时(即应用程序关闭时)，它将检测上面的值是否作过改动；如果是，那么就提醒用户保存信息以备后用：

```
Private Sub Form_QueryUnload(Cancel As Integer,UnloadMode As Integer)
Dim Msg As String
Dim Result As Integer
If txt CompanyName. Text 〈〉 txt Comp anyName. Tag Then
Msg = "You' ve changed the Company Name on the Versions page to"_
txt_CompanyName. Text+_
"Do you want to make the new value your default?"
        Result=MsgBox(Msg,vb YesNoCancel+vbQuestion)
        If Result=vbCancel Then
            Cancel—True
            Exit Sub
        ElseIfResult=vbYes Then
            IniDataL. KeyName= "Company Name"
            IniDatal. Text=txt_CompanyName. Text
        End If
End If
If txt_Author. Text 〈〉 txt_Author. Tag Then
        Msg— "You' ve changed the Author on the Versions page to" +_
          txt_Author. Text+_
          "Do you want to make the new value your default?"
        Result=MsgBox(Msg,vbYesNoCancel+ vbQuestion)
        If Result=vbCancel Then
            Cancel=True
            Exit Sub
        ElseIfResult=vbYes Then
            IniDatal. KeyName- "Author"
            IniDatal. Text=txt_Author. Text
            End If
End If
End Sub
```

这个函数用到了VB4的两个新功能。第一，(终于!)有了续行符。正如我们在书本和杂志文章中一直用的那样，这里的续行符也是一个空格后面跟一条下划线。第二，以往要使用MsgBox函数就必须要包括CONSTANT. TXT里的常量，如今不必这样了。所有常量都由新的浏览器(Browser)管理，不费丝毫力气就可以轻松使用。然而，常量们有了新的名字，因此有必要对旧名字作一次全程查找和替换，例如把MB ICONINFORMATION替换成vbInformation。

如果在表下载时保存了"Author"和"Company Name"信息域中的内容，您会期望在表再次装入的时候检索上次保存的值；事实正如您期望的那样。子过程LoadVersionDefaults在事件Load中调用：

```
Sub LoadVersionDefaults()
IniDatal. Section= "Version"
```

```
IniDatal. KeyName- "Company Name"
txt_CompanyName. Text=IniDatal. Text
txt_CompanyName. Tag=IniDatal. Text
IniDatal. KeyName= "Author"
txt_Author. Text=IniDatal. Text
txt_Author. Tag=IniDatal. Text
txt_ReleaseYear. Text=Format $ (Year(Now))
End Sub
```

记住：文本框内容不论是用程序改变的还是手工改变的，都会激活 Change 事件。因此"Copyright Notice"可以自动地、正确地被更新。

8.4 引入 Listable 类

在讲解 Model 页之前，我们需要花一些时间考虑一下以后的工作该怎么做。我们要管理模块标志位。所谓"管理"，就是要能够显示它们，用户要能够选择它们，我们还要能根据被选中的标志派生出可编译的代码来。此外还有窗口类风格标志和窗口风格标志，它们也应该满足上述要求。所有这些标志都有标识符(identifier)和描述(description)；所有这些标志位都能被选中。而且，它们都将出现在列表框中。

假设"描述"的含义还包括"name(名称)"，那么对标志的一套管理要求也同样适用于标准属性和标准事件以及方法！当然它们都有一些"额外"的要求……但是这对我来说只是类的继承而已。

然而 Visual Basic 的类并不支持 C++意义上的继承。由于某种原因，OLE 的设计者不喜欢这种继承。他们更愿意采用一种称为"aggregation"(聚集)的做法，通过属性来实现继承。与真正的继承相比，这种继承并不方便；我们还得编一些"鹦鹉学舌"式的子过程。但是，比起不用任何继承的手段来说，这仍然能够节省代码，而且可以编写出更具鲁棒性的应用程序来。

在这里有两条平行的继承路线。一条是集合类；另外一条是集合类所包含的内容。图8.2就是一幅简化的层次图，有了它，事情会更清楚些。

基集合类叫做 ListablesClass；它所包含的集合元素的基类叫做 ListableClass。这是我们第一次用 VB 类，因此我们要总体性地介绍一下 VB 的类。在 VB4中，所有文件(除了图形方面的)都以文本格式保存(不再有 VB3的二进制选项)。这样，Notepad 或任何类似的 ASCII、ANSI 文本编辑器都可以阅读它们。VB 对象中，不仅仅是表和控件有属性；连代码模块都有了属性。在 VB 中，属性是在 Properties(属性)窗口中设定的；下面是 ASCII 格式的类属性：

```
VERSION 1.0 CLASS
BEGIN
MultiUse=-1'True
END
Attribute VB_Name= "ListablesClass"
Attribute VB_Creatable=False
Attribute VB_Exposed=False
Attribute VB_Description= "Base collection class for Model, Class and Window flags, and
```

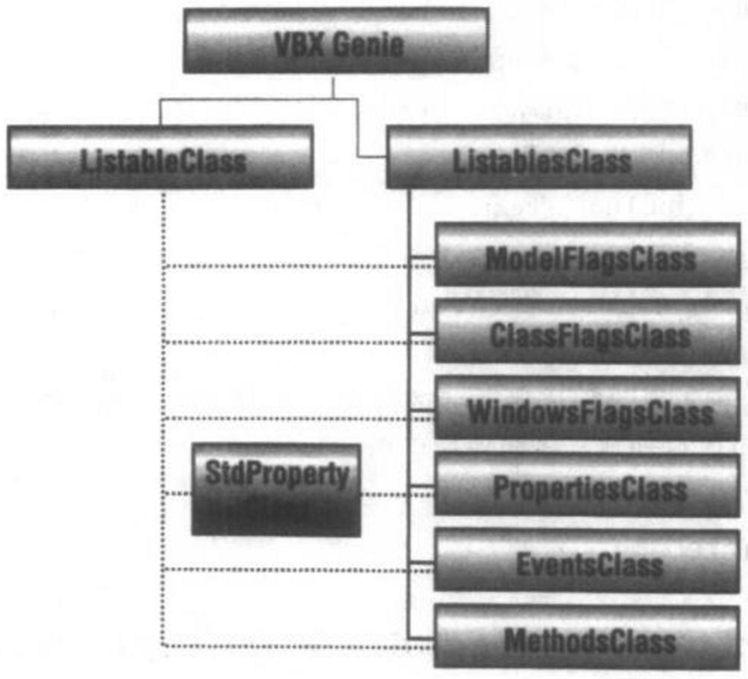

图8.2　显示两条平行继承路线的简化层次图表

properties, events and methods."

VB4可以创建OLE服务器程序；类中的信息大多是用于此目的。然而，VB_Description可以从浏览器中观察到，故而可作为文档的一种形式。(描述内容实际上是在浏览器中设定的。)图8.3给出了浏览器在查看ListablesClass模块时的窗口

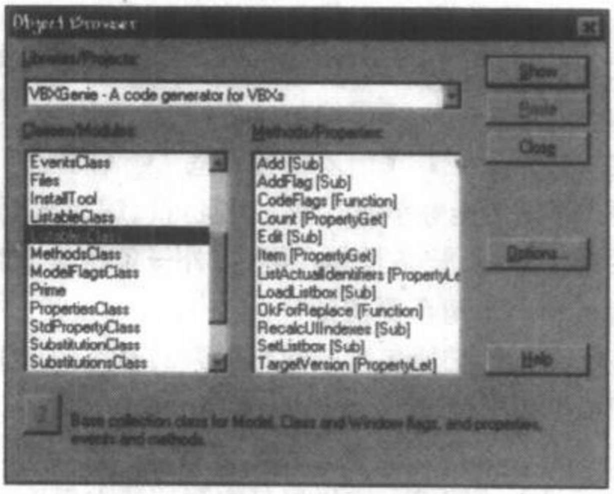

图8.3　VBX Genie的对象浏览器窗口

请不要把模块的属性与类的属性混淆。VB类定义属性的方式类似于控件(另一种VB对象)。

8.4.1　支持集合

ListablesClass就是所谓的"用户自定义集合类"。Visual Basic提供了一个很好的Collection对象，一般可以直接使用。但是如果您有特殊要求，您得使用"聚集"这件VB所提供的、实现继承的唯一法宝，从Collection派生出新类。也就是说，要使ListablesClass成为"用户自定义的集合"，只需让它至少有一个属性是Collection即可：

```
Option Explicit
Private List As New Collection
Private UI As ListBox
```

这些属性全都标记为 Private(私有)，它们不能在类之外引用。除了 Private，还有一种可能情况是 Public(公有)。

关键字 Dim 仍然是可用的，但是在类或表中它的意思是“私有的”，而在代码模块中它的意思是“公有的”。我建议您把意思明确地表达出来，而不要依赖那些会因使用环境不同而发生变化的“默认”。

如何选用这两种存取范围，这与 C++不太一样。在 C++中，属性一般来说都最好是私有的。而在 VB 中，如果某个属性需要保护——即如果您想限制对它的赋值——您就可以把它标记为 Private。除此之外，如果您希望给属性赋值时或者读取属性值时触发某个动作，也可以把它标记为 Private。在没有上述情况需要考虑时，可以安全地把它标记为 Public。

VB 的 Collection 类有一个 Add 方法，因此 ListablesClass 也应该有一个。我们建立了内部的恢复机制，以防止加入关键字相同的项：

```
Public Sub Add(Listable As Object)
On Error Resume Next
List Add Listable, Listable Description
IfErr 〈〉 0 Then
    Err=OkForReplace(Listable)
End If
If Err=0 Then
    Listable. SetListbox UI
    Listable. LoadListbox Index(Listable. Description)
    RecalcUIIndexes
End If
End Sub
```

这个过程不仅仅限于加入 ListableClass 类对象_VB 的集合可容纳任何对象，甚至任何事物。处理重复项的加入是由函数 OkForReplacc 完成的：

```
Private Function OkForReplace(Listable As Object)
OkForReplace-False
Dim Response As Integer
Response=MsgBox("You already have a"+
    Listable. KindOf+_
    "by the name"+_
    Listable. Description+_
    "'. Do you want to replace it?",_
    vb Yes No+vbQuestion,_
    "Duplicate"+Listable. KindOf)
    If Response=VbYes Then
  If Not List(Listable. Description). Standard Then
```

```
        Unload List(Listable. Description)
      End If
      List. Remove Listable. Description
      List. Add Listable, Listable. Description
      OkForReplace=True
    End If
    End Function
```

对象加入之后，对象与用户界面元素列表框的联系就确定下来，并且对象的 LoadListbox 过程被调用。ListablesClass 集合里的所有元素都将与列表框之间建立双向连接。元素自己知道它们在列表框中所处的位置，这由属性 UIIndex 表示。同时，列表框的 ItemData 数组包含了指向 ListablesClass 集合的下标指针。

8.4.2 引入属性函数

我们必须给 ListablesClass 这个用户自定义集合类提供方法，使之尽可能与标准的 VB Collection 类相似，当然除“相似”之外，ListablesClass 还有其他功能。为了要相似，就需要提供“学舌”的辅助件过程。然而，学舌过程与您先前所见到的 VB 过程看起来都有相当显著的差异：

```
Public Property Get Count()As Integer
Count=List Count
End Property
Public Property Get Item(ID As Variant)As Object
Set Item=List. Item(ID)
End Property
```

这是一种全新的处理方法，类似于 C++中的赋值操作符和类型强制转换操作符的重载。VB4把这种称为“属性函数”的方法用来设定属性值和取得属性值。属性函数在类之外看起来与一般的属性并无二致，可出现在等号的两边参与运算。而在属性函数之内，则可以按需加入或多或少的代码。属性函数可用于 Let，Get 和 Set。其中 Let 用于普通变量；Set 用于对象变量。Get 两者都适用。

创建用户自定义集合时会遇到一个令人头疼的问题，上面的 Item 方法即是一例。在 VB 固有的 Collection 类中，Item 是默认的方法。也即是说，下面两条语句是等效的：

```
a=List(6)
a=List. Item(6)
```

我们当然希望我们的用户自定义集合也能这样，但却不行。Visual Basic 并没有提供什么途径把对象的某个方法指定为默认方法，所以在使用时您不得不把它一字不差地拼写出来。

用户自定义集合可以提供给用户统一操作集合元素的方法。比如，假设要对集合里的所有元素都设定某个标志或属性。可以编写属性过程，列举出集合所包含的元素：

```
Public Property Let LisActualIdentifiers(ByVal Value As Boolean)
Dim i As Integer
For i=1 To List. Count
        List(i). ListActualIdentifiers=Value
```

```
Next i
End Property
Public Sub SetListbox(L As ListBox)
Set UI=L
Dim i As Integer
For i=1 To List. Count
    List(i). SetListbox L
Next i
End Sub
Public Property Let TargetVersion(By Val Value As Integer)
Dim i As Integer
For i=1 To List. Count
    List(i). TargetVersion=Value
Next i
End Property
```

按惯例，在对象类型的上下文范围内，一种方法的名称应该能够准确体现此方法所完成的工作内容。由于这个惯例，ListablesClass 类就有了命名为 LoadListbox 的方法，它装入一个列表框的各项内容：

```
Public Sub LoadListbox()
Dim PreviousSelection As Integer
IfUI. ListCount>0 And UI. ListIndex>-1 Then
    PreviousSelection=UI. ItemData(UI.ListIndex)
Else
    PreviousSelection=-1
End If
UI. Clear
Dim i As Integer
For i=1 To List. Count
    List(i). LoadListbox i
Next i
RecalcUIIndexes PreviousSelection
End Sub
Private Sub RecalcUIIndexes(Optional PreviousSelection As Varant)
Dim i As Integer
If IsMissing(PreviousSelection)Then
    PreviousSelection=-1
End If
For i=0To UI. ListCount-1
    List. Item(UI. ItemData(i). UIIndex=i
    If UI. ItemData(i)=Val(PreviousSelection)Then
```

```
        UI. List Index-i
        UI. TopIndex=i
      End If
    Nexti
    End Sub
```

RecalcUIIndexes 被单独作为一个过程来编写，因此可以独立于 LoadListbox 调用，比如加入集合元素时就会用到 RecalcUIIndexes。(见前面的 Add 方法。)

还有，请不要错过 VB4的另一个新特色：可选参数。过程的某些或所有的参数都可以是可选的。可选参数必须是 variant 类型，并且在参数表中可选参数的后面不跟有别的参数。内部的 IsMissing 函数判断是否给出了可选参数。

8.4.3 完成初始化

ListablesClass 类还包含了一些其他的方法，我准备在用到它们的时候再作解释。现在，我们看看 ListablesClass 包含的元素：ListableClass。

ListableClass 类对象应该具有标识符、描述内容，并可以选中。在它的(General)(Declarations)中可见到支持上述特征的属性：

```
Option Explicit
Public Identifier As String
Public Description As String
Private Selected_ As Boolean
Private Version As Integer
Public ListActualIdentifiers As Boolean
Public TargetVersion As Integer
Private UI As Control
Public UIIndex As Integer
```

尽管应用程序的其他部分也可能创建 ListableClass 对象并手工设定其属性，但是调用一个简单的过程 Init 完成所有的初始化工作会更加容易些：

```
Public Sub Init(aIdentifier As String, _
                aDescription As String, _
                ByVal aVersion As Integer)
Identifier=aIdentifier
Description=aDescription
Version=aVersion
End Sub
```

这样做还有助于实现数据的隐藏，因为有些属性是私有的。

下面是 ListableClass 类的 SetListbox 函数，它将被 ListablexClass 类的 SetListbox 函数所调用：

```
Public Sub SetListbox(L As ListBox)
Set UI=L
End Sub
```

并不是所有的属性过程都必须引用内存中的变量。举例来说，当 ListableClass 类对象用作标准的属性、事件或方法的时候，对象的 Standard 属性必须返回常量 True；而用作用户属性和事件时，Standard 属性返回常量 False。Standard 属性是用代码实现的，而不是变量：

```
Public Property Get Standard()As Boolean
Standard=True
End Property
```

如果不提供 Property Let 函数，我们就可以使 Standard 成为“只读”的属性。很妙吧?

对于 Selected 属性来说，我们需要存储一个数值，即 Selected_这个私有变量的值。这是 Get 函数所完成的工作，与 Standard 类似：

```
Public Property Get Selected()As Boolean
Selected=Selected_
End Property
```

为什么要用属性函数实现 Selected?这是因为 Let 函数还有其他事情要做：

```
Public Property Let Selected(By Val Value As Boolean)
Selected_=Value
IfNot (UI Is Nothing)Then
    UI. Selected(UIIndex)=Value
End If
End Property
```

我们希望对象与它们各自的列表框相联系。这样在标志、属性、事件或方法被选中时，相应的列表项就反映出它已被选中。如果对象自己做到这一点，我们就不必操心了。

剩下的两个属性 ListActualldentifiers 和 TangetVersion 都是公有的，它们在 ListablesClass 的属性过程中设定，这您已经见过了。前一个属性是 Boolean 型变量，它通知本对象在列表框中要显示的是 Identifer 字段还是 Description 字段。第二个属性的作用是如果要创建的目标 VBX 的版本低于对象自己的标志、属性、事件或方法，那么对象可以把自己从列表框中滤除。

这都在类的 LoadListbox 方法中得以实现：

```
Public Sub LoadListbox(By Val Index As Integer)
  If Version<=TargetVersion Then
  If ListActualldentifiers Then
      UI. AddItem Identifier
  Else
      UI. AddItem Description
  End If
  UI. ItemData(UI. Newindex)=Index
  UI. Selected(UI. Newindex)=Selected—
End If
End Sub
```

可见，列表项的 ItemData 属性被设定为该项的索引(由调用者传递)。这样每个列表项都可以访问与之关联的 ListableClass 对象。属性 UIIndex 给出了从 ListableClass 对象到列表项的反向连接。(UIIndex 是在所有的列表项都装入以后，由 LoadListbox 的调用者设定。)

注意，多项选择列表框中的各项可根据自己是否是“选中的”(即它的 Selected 属性是否为真)而在框中被预先选中。

您也许不明白，为什么要不厌其烦地把模块标志、类风格标志、窗口风格标志分门别类？这些细化了的类还有什么特长？答案在于：它们可以定义出适当的标志集合。

8.5 支持 Model 页

在 VBX 中，MODEL 结构把一个给定控件的所有代码和数据联结在一起。因此，Model 页是 VBX Genie 中最为复杂的部分也就不足为怪了。它包含有自己的标签页！图8.4是显示模块标志时的外观。

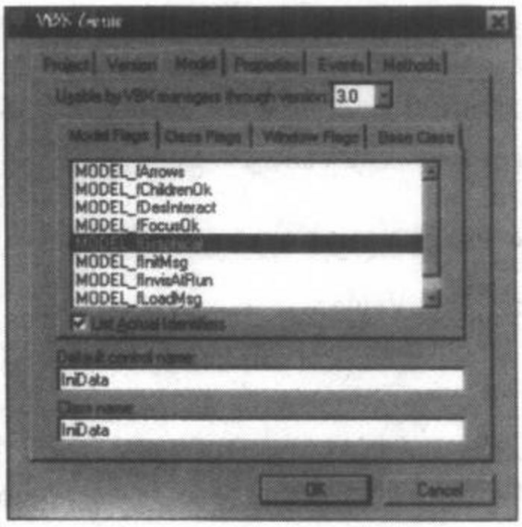

图8.4 VBX Genie 的模块标志

除了模块标志，本页还提供了对窗口类风格标志和窗口风格标志的存取能力。这三个子页布局都很相似，而且它们都由我们刚刚讨论过的 ListablesClass 类管理。

模块标志直接由 ModelFlagsClass 类管理，该类是从 ListablesClass 类派生的——哦，对不起，是“聚集”。这个标志集合是所有三个标志位集合中最复杂的，只有它具有某些标志和标准属性之间复杂的交互作用。这种复杂性在(General)(Declarations)中有所反映：

```
Option Explicit


Public Base As New ListablesClass
Private Properties As PropertiesClass
```

由于模块标志与标准属性之间的交互作用，所以这里我们有一个指向 PropertiesClass 对象的指针。(就其本身来说，它是一个指针，而不是一个对象，因为 Private 语句当中没有关键字 New。)

Flags 成员不用我多费口舌；只提醒您注意这个类是从 ListablesClass 类派生”的(通过聚集)。那么这些标志是在哪里定义的呢？它们是在初始化过程中定义的。然而，当代码窗口的“object”组合框选中“(General)”并拉下“Proc”组合框时，您见不到这个初始化过程。可见，除了属性外，VB 用户自定义类还有 constructors(构造函数)和 destructors(析构函数)。它们被

称为“初始化函数(initializer)”和“终止函数(terminators)”，这只是术语上的区别。

initializers 和 termmators 很容易被忽略，它们不是(General)对象的一部分；它们属于模块的 Class 对象。

象 Flags 这样的集合属性，初始化函数是设置它们的最佳场所。

```
Private Sub Class Initialize()
Base. AddFlag" MODEL_fArrows"," Require arrow keys at run_time", 1
Base. AddFlag" MODEL_fChildrenOk"," Child control container", 1
Base. AddFlag" MODEL_fDesInteract"," Receive right mouse button messages at design
time", 1
Base. AddFlag" MODEL_fFocusOk"," Can receive focus at run-time", 1
Base. AddFlag" MODEL_fGraplncal"," Graphical control", 2
Base. AddFlag" MODEL_flnitMsg"," Receive VBM_INITIALIZE messages", 1
Base. AddFlag" MODEL_flnvisAtRun"," lnvisible at run-time", 2
Base. AddFlag" MODEL_fLoadMsg",_
    " Receive VBM CREATED and VBM LOADED messages", 1
Base. AddFlag" MODEL_fMnemonic"," Respond to hot keys", 1
End Sub
```

在(General)里面列出的过程中有很多名字很面熟。例如 Selected 就列在其中。您可能会设想这个 Selected 必须调用基类的 Selected，这个设想当然没错，然而它还得管理我所说的那些交互作用：

```
Public Property Let Selected(Index As Variant,ByVal Value As Boolean)
Static Busy As Boolean
IfBusy Then Exit Properly
Busy=True
Base. Item(Index). Selected=Value
Select Case Index
Case" MODEL_flnvisAtRun"
    IfValue Then
        Properties. Selected("Visible")=False
    End If
Case" MODEL_fGraphical"
    IfValue Then
        Properties. Selected("hWnd")=False
    End If
End Select
Busy=False
End Property
```

但是，类中的大部分过程都只是对应 ListablesClass 类过程的简单重复：

```
Public Property Let ListActualIdentifiers(ByVal aValue As Boolean)
Base. LjstActualIdentifiers=aValue
End Properly

Public Sub LoadListbox()
Base. LoadListbox
End Sub

Public Property Let TargetVersion(ByVal Value As Integer)
Base. TargetVersion=Value
End Property
```

另外两个 VBX Genie 标志类的设计思想与此类似。例如，它们都有一个 Listableclass 类型的 Flags 属性。由于没有交互作用要考虑，ClassFlagsClass 和 indowFlagsClass 比 ModelFlagsClass 类更简单。那么，它们的用处何在呢?它们可以把一套适当的标志收集到自己的集合中。下面是 ClassFlagsClass 类的初始化函数：

```
Private Sub Class_Initialize()
Base. AddFlag" CS_BYTEALIGNCLIENT"," Align x-axis on byle boundary", 0
Base. AddFlag" CS_CLASSDC"," Share device context with other controls of this class",
0
Base. AddFlag" CS_DBLCLKS"," Receive double click messages", 0
Base. AddFlag" CS_HREDRAW"," Redraw if horizontal size changes", 0
Base. AddFlag" CS_OWNDC"," Private device context", 0
Base. AddFlag" CS_VREDRAW"," Redraw if vertical size changes", 0
Base. AddFlag" CS_SAVEBITS"," Minimize redraws when possible", 0
End Sub
```

Add 的第三个参数是最先支持本标志位的 VBX 管理器的版本。当然，这只是对模块标志才有意义；窗口类风格标志是 Windows 自带的，比 Visual Basic 的第一个版本都要早得多. 因此它们的“版本”值被设置为0。WindowFlagsClass 类也是这样：

```
Private Sub Class_Initialize()
Base. AddFlag" WS_CLIPCHILDREN"," Don't erase contained child controls", 0
Base. AddFlag" WS_CLIPSIBLINGS"," Don't erase overlapping controls", 0
Base. AddFlag" WS_GROUP"," First control of group navigable via arrow keys", 0
Base. AddFlag" WS_HSCROLL"," Has horizontal scroll bar", 0
Base. AddFlag" WS_TABSTOP"," Can tab to this control", 0
Base. AddFlag" WS_VSCROLL"," Has vertical scroll bar", 0
End Sub
```

有的窗口风格标志并不适用于 VBX 控件，比如 WS_OVERLAPPED。这样的标志位就不

列入了。

在表一级，用户可以通过三种途径与标志类交互。第一，单击版本组合框(从 Version 页)。这样会触发重新装入所有列表框，代码我们已经见过了。第二，用户可以选中或不选“List Actual Identifiers”复选框：

```
Private Sub chK_ListModelIdentifiers_Click()
ModelFlags. ListActualIdentifiers=(chk_ListModelIdentifiers=vbChecked)
ModelFlags. LoadListbox
End Sub

Private Sub chk_ListClassIdentifiers_Click()
ClassFlags. ListActualIdentifiers=(chk_ListClassIdentifiers=vbChecked)
ClassFlags. LoadListbox
End Sub

Private Sub chk_ListWindowIdentifiers_Click()
WindowFlags. ListActualIdentifiers=(chk_LisWindowIdentifiers=vbChecked)
WindowFlags. LoadListbox
End Sub
```

最后一种途径，用户可以单击标志项：

```
Private Sub lst_ModelFlags_Click()
If Screen. ActiveControl Is lst_ModelFlags Then
    Dim i As Integer
    i=lst_ModelFlags. ListIndex
    ModelFlags. Selected(lst_ModelFlags. ItemData(i)) =lst_ModelFlags. Selected(i)
End If
End Sub

Private Sub lst_ClassFlags_Click()
If Screen. ActiveControl Is lst_ClassFlags Then
    Dim i As Integer
    i=lst_ClassFlags. ListIndex
    ClassFlags. Selected(lst_ClassFlags. ItemData(i)) =lst_ClassFlags. Selected(i)
End If
End Sub

Private Sub ist_WindowFlags_Click()
If Screen. ActiveControl Is lst_WindowFlags Then
    Dim i As Integer
    i=lst_WindowFlags. ListIndex
```

```
        WindowFlags. Selected(lst_WindowFlags. ItemData(i)) = lst_WindowFlags. Selected
(i)
    End If
    End Sub
```

比方说，用户单击了 MODEL_fGraphical 项。列表框的 Click 事件将被触发，ModelFlags 对象的 Selected 属性函数标记出这个标志已被选中，同时还保证 hwnd 标准属性不会被选中。

8.5.1 Model 页的最终实现

这里作一个扼要的重述。我们已经有了 ListablesClass 类，它容纳的全都是 ListableClass 对象；同时，ListablesClass 类是 ModelFlagsClass 类、ClassFlagsClass 类、WindowFlagsClass 类的基础，这些类分别用于模块标志、窗口类风格标志、窗口风格标志。

如果您是初涉面向对象程序设计，您可能觉得这种事无巨细都要过问的方式太费事了！但是请想一想：这些函数只有很少的代码，因此不大容易出错。即使不小心程序出了错，也可以很容易地发现并改正过来，因为各个函数的代码都是非常简单的。复杂的是关系，而如果我们把话题限制在程序设计而不是人上，那么关系可以轻易地理顺！

Model 页上还有一个标记为“Base Class”的标签。VBX 可基于一个已有的 Windows 类；也可基于您编写的用户控件。Base Class 标签上有一个简单的组合框，其中列出 VB4预先支持的标准 Windows 控件——可在设计的时候完成！。按下 OK 按钮后，读入用户的选择，对此我们以后再作考虑。

8.6 支持 Properties 页

Prime 表的下一个标签涉及到控件中用到的所有标准的和用户的属性。Properties 页外观如图8.5所示。

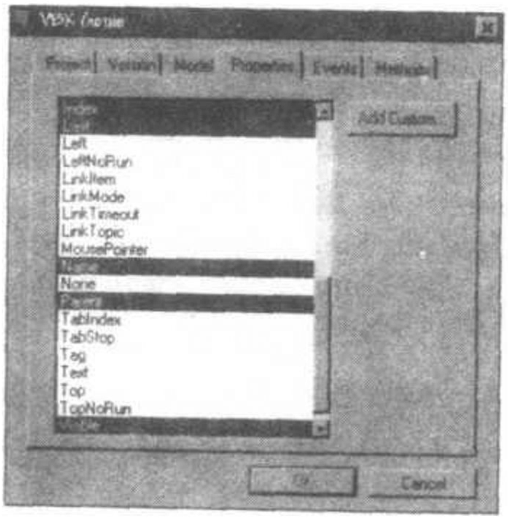

图8.5 VBX Genie 的 Properties 页

Name、Index、Parent 三个必需属性以及少量常用的标准属性，如 Enabled，都已经预先选定。这是一个多选列表框；用户要选中一项，只需简单地单击它，再次单击可以取消选中。

有些属性是"组"属性。例如，当选定一个与字体有关的属性时，所有与字体有关的属性都必须被选中。与连接和拖动有关的属性也是组属性。成组的过程应该自动完成，我们在实现本页时要考虑到这一点。还有些属性要求与某些 MODEL 标志共存，而有些则与 MODEL 标志是互斥的，共存与互斥也需要自动实现。

您在列表框中所见的项都是标准属性。尽管我们也要支持用户属性，但是本列表框内的各项，不论选中还是未选中，大都是标准的属性。

由于标准属性和 MODEL 类标志之间存在着交互作用，所以不能直接把 ListableClass 对象当成标准属性来用。这样我们不得不定义一个 StdPropertyClass 类。下面是 StdPropertyClass 类的(General)(Declarations)部分：

```
Option Explicit


Private Base As New ListableClass
Public ModelFlags As ModelflagsClass
Public Required As Boolean
```

StdPropertyClass 类也是从 ListableClass 聚集而成的。我们已经知道 ListableClass 的工作原理。那么 StdPropertyClass 除此之外又加进了些什么呢?

加入了 ModelFlags 指针：这显然是服务于标准属性和 MODEL 标志间的交互作用的。还加入了 Required。VBX 标志、事件、方法都不是必需的；所以 Required 标志是这个类的特色。标准属性在放入属性列表框的时候就可以被预先选中；例如，Name 和 Index 这样的必需属性就已经预先选中了。这些都在 Init 方法里处理，处理之前应先调用基类的 Init 方法：

```
Public Sub Init(aIdentifier As String aDescription As String ByVal aVersion As Integer, _
        ByVal aRequired As Boolean, ByVal aSelected As Boolean)
Base. Init aIdentifier, aDescription, aVersion
Required=aRequired
Selected=aSelected Or aRequired
End Sub
```

由于用户可能会想取消选中某个必需的标准属性，而我们又不允许用户那样做。Property Let 函数 Selected 处理这一特殊情况：

```
Public Property Let Selected(ByVal Value As Boolean)
If Required And Value=False Then
    Base. Selected=True
    MsgBox Base. Description+" is a required property.", vbExclamation," properties"
Else
    Base. Selected=Value
Else If
End Property
```

如果用户取消选中一个必需项，本函数在显示消息框之前又把它重新选中。(如果消息框

清除之后再重新选中，这看起来似乎是在和用户对着干。）

StdPropertyClass 的其余方法都是基类方法的重复，这里就不再浪费时间介绍了。

另一个用户定义集合 PropertiesClass 容纳所有这些标准属性对象。下面是它的(General)(Declarations)部分：

```
Option Explicit

Public Base As New ListablesClass
Private ModelFlags As ModelFlagsClass
Private Events As EventsClass
```

跟上文的标志类一样，PropertiesClass 在初始化时创建出一系列标准属性项：

```
Private Sub Class_initialize()
" Name" and" Index" MUST be the first two items in this list
AddStdProperty" PPROPINFO_STD_NAME"," Name", 1, True, True
AddStdProperty" PPROPINFO_STD_INDEX"," Index", 1, True, True
AddStdProperty" PPROPINFO_STD_ALIGN"," Align", 2, False, False
AddStdProperty" PPROPINFO_STD_BACKCOLOR"," BackColor", 1, False, False
AddStdProperty" PPROPINFO_STD_BORDERSTYLEOFF"," BorderStyle", 1, False,
False
AddStdProperty" PPROPINFO_STD_CAPTION"," Caption", 1, False, False
AddStdProperty" PPROPINFO_STD_CLIPCONTROLS"," ClipControls", 2, False, False
AddStdProperty" PPROPINFO_STD_DATACHANGED"," DataChanged", 3, False, False
AddStdProperty" PPROPINFO_STD_DATAFIELD"," DataField", 3, False, False
AddStdProperty" PPROPINFO_STD_DATASOURCE"," DataSource", 3, False, False
AddStdProperty" PPROPINFO_STD_DRAGICON"," DragIcon", 1, False, False
AddStdProperty" PPROPINFO_STD_DRAGMODE"," DragMode", 1, False, False
AddStdProperty" PPROPINFO_STD_ENABLED"," Enabled", 1, False, True
AddStdProperty" PPROPINFO_STD_FONTBOLD"," FontBold", 1, False, False
AddStdProperty" PPROPINFO_STD_FONTITALIC"," FontItalic", 1, False, False
AddStdProperty" PPROPINFO_STD_FONTNAME"," FontName", 1, False, False
AddStdProperty" PPROPINFO_STD_FONTSIZE"," FontSize", 1, False, False
AddStdProperty" PPROPINFO_STD_FONTSTRIKE"," FontStrike", 1, False, False
AddStdProperty" PPROPINFO_STD_FONTUNDER"," FontUnder", 1, False, False
AddStdProperty" PPROPINFO_STD_FORECOLOR"," ForeColor", 1, False, False
AddStdProperty" PPROPINFO_STD_HELPCONTEXTID"," HelpContextID", 2, False,
False
AddStdProperty" PPROPINFO_STD_HWND"," hWnd", 1, False, False
AddStdProperty" PPROPINFO_STD_IMEMODE"," ImeMode", 2, False, False
AddStdProperty" PPROPINFO_STD_TOPNORUN"," TopNoRun", 2, False, False
AddStdProperty" PPROPINFO_STD_LEFTNORUN"," LeftNoRun", 2, False, False
```

```
AddStdProperty" PPROPINFO_STD_TOP"," Top", 1, False, False
AddStdProperty" PPROPINFO_STD_LEFT"," Left", 1, False, False
AddStdProperty" PPROPINFO_STD_HEIGHT"," Height", 1, False, False
AddStdProperty" PPROPINFO_STD_WIDTH"," Width", 1, False, False
AddStdProperty" PPROPINFO_STD_LINKITEM"," LinkItem", 2, False, False
AddStdProperty" PPROPINFO_STD_LINKMODE"," LinkMode", 2, False, False
AddStdProperty" PPROPINFO_STD_LINKTIMEOUT"," LinkTimeout", 2, False, False
AddStdProperty" PPROPINFO_STD_LINKTOPIC"," LinkTopic", 2, False, False
AddStdProperty"PPROPINFO_STD_MOUSEPOINTER","MousePointer",1,False,False
AddStdProperty" PPROPINFO_STD_NONE"," None", 2, False, False
AddStdProperty" PPROPINFO_STD_PARENT"," Parent", 1, True, True
AddStdProperty" PPROPINFO_STD_TABINDEX"," TabIndex", 1, False, False
AddStdProperty" PPROPINFO_STD_TABSTOP"," TabStop", 1, False, False
AddStdProperty" PPROPINFO_STD_TAG"," Tag", 1, False, False
AddStdProperty" PPROPINFO_STD_TEXT"," Text", 1, False, False
AddStdProperty" PPROPINFO_STD_VISIBLE"," Visible", 1, False, True
`" Last" MUST be the final item in this list
AddStdProperty" PPROPINFO_STD_LAST"," Last", 2, False, False
End Sub
```

每个标准属性都是 StdPropertyClass 类的对象，对象在 AddStdProperty 方法中被实例化并成为集合的一部分：

```
Private Sub AddStdProperty(Identifier As String,Descnption As String,
        ByVal Version As Integer, ByVal Required As Integer, ByVal Selected As
        Integer)
Dim Item As New StdPropertyClass
Item. Init Identifier, Description, Version, Required, Selected
Base. Add Item
End Sub
```

当某个属性被选中(或取消选中)时，接下来就要处理属性之间的交互作用以及属性与标志和事件之间的交互作用了：

```
Public Property Let Selected(ByVal ID As String,ByVal Value As Boolean)
Static Busy As Boolean
If Busy Then Exit Property

Busy=True
If Left$(ID,4)=" Data" Then
    Base. Item("DataChanged"). Selected=Value
    Base. Item("DataField"). Selected=Value
    Base. Item("DataSource"). Selected=Value
```

```
ElseIf Left $ (ID,4)=" Drag" Then
    Base. Item("DragIcon"). Selected=Value
    Base. Item("DragMode"). Selected=Value
    Events. Selected("Drag")=Value
ElseIf Left $ (ID,4)=" Font" Then
        Base. Item("FontBold"). Selected=Value
        Base. Item("FontItalic"). Selected=Value
        Base. Item("FontName"). Selected=Value
        Base. Item("FontSize"). Selected=Value
        Base. Item("FontStrike"). Selected=Value
        Base. Item("FontUnder"). Selected=Value
ElseIf Left $ (ID,4)=" Link" Then
        Base. Item("LinkItem"). Selected=Value
        Base. Item("LinkMode"). Selected=Value
        Base. Item("LinkTimeout"). Selected=Value
        Base. Item("LinkTopic"). Selected=Value
        Events. Selected("Link")=Value
Else
        Base. Item(ID). Selected=Value
        IfValue Then
                Select Case ID
                Case" Caption"
                        Base. Item("Text"). Selected=False
                Case" ClipControls"
                        ModelFlags. Selected("MODEL_fChildrenOK")=True
                Case" hWnd"
                        ModelFlags. Selected("MODEL_fGraphical")=False
                Case" TopNoRun"," LeftNoRun"
                        ModelFlags. Selected("MODEL_finvisAtRun")=True
                        Base. Item("Top"). Selected=False
                        Base. Item("Left"). Selected=False
                        Base. Item("Height"). Selected=False
                        Base. Item("Width"). Selected=False
                        Base. Item("TopNoRun"). Selected=True
                        Base. Item("LeftNoRun"). Selected=True
                Case" LinkItem"," LinkMode"," LinkTimeout"," LinkTopic"
                        ModelFlags. Selected("MODEL_fLoadMsg")=True
                Case" Text"
                        Base. Item("Caption"). Selected=False
                Case" Top"," Lsft"," Height"," Width"," Visible"
```

```
                ModelFlags. Selected("MODEL_finvisAlRun")=False
                Base. Item("Top"). Selected=True
                Base. Item("Left"). Selected=True
                Base. Item("Height"). Selected=True
                Base. Item("Width"). Selected=True
                Base. Item("TopNoRun"). Selected=False
                Base. Item("LeftNoRun"). Selected=False
            End Select
        End If
    End If
    Busy=False
    End Property
```

要是您能够理解 Visual Basic 语法的话，各种相互关系其实都已经在这里交代清楚了。

注意静态变量 Busy 的用法。由于存在着交互作用，假如缺乏某种预防机制的话，就会不可避免地发生无穷递归的情况。Busy 标志则提供了这样的预防机制。

您也许还记得，ListableClass 有一个 Standard 属性经常设定为 True。(实际上 StdPropertyClass 也是这样，比起对 Standard 属性的低水平重复，这样做的效率更高。)显然 PropertiesClass 需要存储用户属性，就象存储标准属性那样。这够得上用一个小节的内容来阐述。

8.7 支持用户属性

VBX 控件之所以有用，主要原因还在于用户属性。然而用户属性有太多的情况要处理！要处理不同的数据类型，比如包含了数值列表的“枚举”类型，还要处理许多存储选项。按动“Add Custon…”按钮时，Property 页将显示出一个对话框，见图8.6，设计这个对话框就是要解决上述复杂问题。

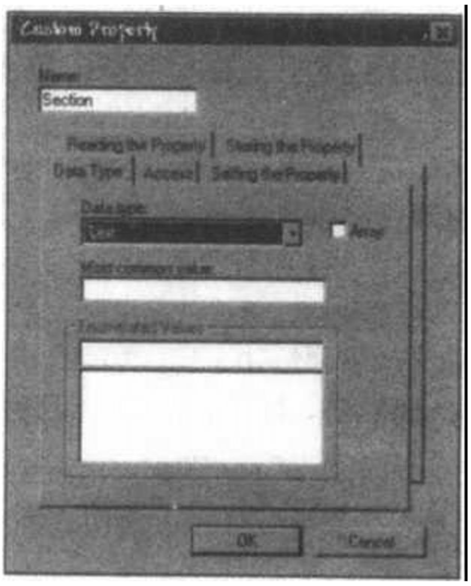

图8.6 组织用户属性的 Custom Property 对话框

本表有五个标签页，然而大部分工作都在第一页完成。用户在第一页指定用户属性的数据类型。如果是枚举类型，就可以指定其枚举项。如果用户属性经常要用到某个值，也可以指明它，这样 VBX 管理器就不必存储它了。

Visual Basic 一直都有表，并且从 VB2开始，表被当成是用户自定义的对象，但是由于表对象没有公有“属性”和“函数”，因而不能发挥出更大作用。而在 VB4中，表就成了有脸有面的类。因此，可以把用户属性类实现为表，这样就避免了把数据对象与用户界面元素相对应的麻烦。因为对应关系是一目了然的。

我本来想把新类命名为“CustomPropertyClass”，但在最后一刻放弃了这个打算，把它改为 CustomPropertyForm，我害怕表和类区别不开(即使它们在 Project 窗口里用不同的图标表示)。在(General)(Declarations)中有下列属性：

```
Option Explicit

Public OK As Boolean

Public Description As String
Public Selected_As Integer

Private DataType As String
Private DefaultValue As String
Private Ennmerations As New Collection
Public Flags As New ListablesClass

Public UI As ListBox
Public UIIndex As Integer
```

对象的拥有者利用 OK 属性判定用户是否敲击了 OK 按钮。Description 和 Selected_不用多费口舌。DataType、DefaultValue、Enumerations 和 Listables 的用处是显而易见的；我一会儿再作细致地介绍。UI 和 UIIndex 这两个属性您在 ListableClass 类中就很熟悉了。

我早先曾经提到过，表和类一样，都有一个 Initialize 事件。这个事件在 Load 事件之前发生；当 Initialize 事件进行时，表还没有在内存里建立起来，表的控件也没有准备就绪。因此初始化只是为 Flags 这样的非控件属性提供舞台：

```
Private Sub Form_Initialize()
Flags. AddFlag" PF_fGetHsZMsg","", 0
Flags. AddFlag" PF_fPropArray","", 0
Flags. AddFlag" PF_fEditable","", 0
Flags. Item("PF_fEditable"). Selected=True
Flags. AddFlag" PF_fNoMultiSelect","", 0
Flags. AddFlag" PF_fNoShow","", 0
Flags. AddFlag" PF_fUpdateOnEdit","", 0
Flags. AddFlag" PF_fNoRuntimeW","", 0
```

```
Flags. AddFlag" PF_fNoRuntimeR","", 0
Flags. AddFlag" PF_fSetData","", 0
Flags. Item("PF_fSetData"). Selected=True
Flags. AddFlag" PF_fSetMsg","", 0
Flags. AddFlag" PF_fSetCheck","", 0
Flags. AddFlag" PF_fGetData","", 0
Flags. Item("PF_fGetData"). Selected=True
Flags. AddFlag" PF_fGetMsg","", 0
Flags. AddFlag" PF_fSaveData","", 0
Flags. Item("PF_fSaveData"). Selected=True
Flags. AddFlag" PF_fSaveMsg","", 0
Flags. AddFlag" PF_fLoadMsgOnly","", 0
End Sub
```

联想到 VBX 框架结构，您对这些标志应该不会感到陌生；它们都是用来定义用户属性的属性标志。其中有一些标志是预先选定的。它们的值以后将提供给表上的复选框。

比方说，如果用户定义了五个用户属性，在内存中就会有五个这样的表，即使在某个时候只有一个表可见甚至没有一个表可见也是这样。认识到这一点是很重要的。表与数据相联系。因此，用户界面元素的值必须在处理表和 Active 事件时从对象的属性那里获得：

```
Private Sub Form_Activate()
cmd OK. Enabled=VerifyOK
OK=false
txt_Description. Text=Description
cmb_DataType. ListIndex=_
    ConvertDataType(DataType)
chk_PropArray=_
    SetCheck(Flags. Item("PF_fPropArray"). Selected)
txt_CommonValue=DefaultValue
SetEnumerations cmb Eηnmerations
chk_PropertyIISZ=_
    SetChcck(Flags. Item("PF_fGetHszMsg"). Selected)
chk_MultiSelect=SetCheck(Not Flags. Item("PF_fNoMultiSelect"). Selected)
chk_Show=SetCheck(Not Flags. Item("PF_fNoShow"). Selected)
chk_UpdateOnEdit=SetCheck(Flags. Item("PF_fUpdateOnEdit"). Selected)
chk_RunTimeWrite=SetCheck(Not Flags. Item("PF_fNoRuntimeW"). Selected)
chk_RunTimeRead=SetCheck(Not Flags. Item("PF_fNoRuntimeR"). Selected)
opt_SetData=Flags. Item("PF_fSetData"). Selected
opt_SetMsg=Flags. Item("PF_fSetMsg"). Selected
opt_SetMsgData = Flags. Item ( " PF_ fSetData"). Selected And Flags. Item ( " PF_
fSetMsg"). Selected
```

```
chk_SetCheck=SetCheck(Flags.Item("PF_fSetCheck"). Selected)
opt_GetData=Flags. Item("PF_fGetData"). Selected
opt_GetMsg=Flags. Item("PF_fGetMsg"). Selected
opt_SaveData=Flags. Item("PF_fSaveData"). Selected
opt_SaveMsg=Flags. Item("PF_fSaveMsg"). Selected
End Sub
```

之所以要在 Activate 事件中处理，是因为表只能装入一次，而如果用户要改变其初始设置时，表可以激活多次。

您也许会问"如果表是装入的，那为什么不一次填入这些值就不管了呢？"如果没有 Cancel 按钮的话当然可以这样。问题是，应该让用户有推倒重来的机会，而在表激活期间再重新填入各个信息域则为用户提供了这样的机会。用户击中 Cancel 按钮的处理很简单：

```
Private Sub cmd_Cancel_Click()
Hide
End Sub
```

仅当用户单击 OK 按钮后，数据元素才会更新：

```
Private Sub cmd_OK_Click()
Description=txt_Description. Text
DataType=ConvertDataType(cmb_DataType. ListIndex)
Flags. Item("PF_fPropArray"). Selected-chk_PropArray
DefaultValue=txt_CommonValue
GetEnumerations cmb Enumerations
Flags. Item("PF_fGetIlszMsg")" Selected=chk_PropertyIISZ
Flags. Item("PF_fNoMultiSelect"). Selected=Not GetCheck(chk_MultiSelect)
Flags. Item("PF_fNoShow"). Selected=Not GetCheck(chk_Show)
Flags. Item("PF_fUpdateOnEdit"). Selected=GetCheck(chk_UpdateOnEdit)
Flags. Item("PF_fNoRuntimeW"). Selected=Not GetCheck(chk_RunTimeWrite)
Flags. Item("PF_fNoRuntimeR"). Selected=Not GetCheck(chk_RunTuneRead)
Flags. Item("PF_fSetData"). Selected=opt_SetData Or opt_SetMsgData
Flags. Item("PF_fSetMsg"). Selected=opt_SetMsg Or opt_SetMsgData
Flags. Item("PF_fSetCheck"). Selected=GetCheck(chk_SetCheck)
Flags. Item("PF_fGetData"). Selected=opt_GetData
Flags. Item("PF_fGetMsg"). Selected=opt_GetMsg
Flags. Item("PF_fSaveData"). Selected=opt_SaveData
Flags. Item("PF_fSaveMsg"). Selected=opt_SaveMsg
OK-True
Hide
End Sub
```

VB 表的这些小把戏我就不再多说。但我有个新思想要告诉您。过去我设计的组合框通常

都带有命令按钮，用它来添加新项或删除已有项。这样的布局对用户来说当然是再清楚不过了，但是如果要添加几项而不得不按动按钮，这毕竟是件烦人的事。所以这次我作了些小改动，希望用户能明白。

仅当数据类型是枚举型时，用户属性表上的组合框才是激活的。这样，用户可以输入任意多个数值：这些值将与属性一起存储，并且在控件的 VBX 管理器属性窗口中出现。在用户属性表上，如果在组合框的编辑部分输入数值，那么输入焦点移动到其他信息域后，数值将自动加入到列表中。这是在组合框的 LostFocus 事件中完成的：

```
Private Sub cmb Enumerations_LostFocus()
Dim i As Integer
i=cmb_Enumerations. ListIndex
if cmb_Enumerations. Text)"" And cmb_Enumerations. Text <> cmb_Enumerations. List
(i) Then
        cmb_Enumerations. AddItem cmb_Enumerations. Text
        cmb_Enumerations. Text-""
        cmb_OK. Enabled=VerifyOK
End If
End Sub
```

此外用户敲击 Insert 键也可获得同样效果。而且，如果编辑区的文本与当前选中的项匹配-用户用鼠标单击一个列表项就是这样-用户可敲击 Delete 键删除这一项：

```
Private Sub cmb_Enumerations_KeyDown(KeyCode As Integer,Shift As Integer)
Dim Temp As String i As Integer
i=cmb_Enumerations. ListIndex

IfKeyCode=vbKeyDelete Then
        If cmb_Enumerations. Text=cmb_Enumerations. List(i)Then
                cmb_Enumerations. RemoveItem i
                cmd_OK. Enabled=VerifyOK
        End If

ElseIfKeyCode=vbKeyInsert Then
        cmb_Enumerations. AddItem cmb_Enumerations. Text
        cmb_Enumerations. Text=""
        cmd_OK. Enabled=VerifyOK

ElseIfKeyCode=vbKeyUp And((Shift And vbShiftMask)>0) Then
        Ifi) 0 Then
                Temp-cmb_Enumerations. List(i)
                cmb_Enumerations. RemoveItem i
                cmb_Enumerations. AddItem Temp, i-1
```

```
                cmb_Enumerations. ListIndex=i-1
                KeyCode=0
        Else
                Beep
        End If

    ElseIfKeyCode=vbKeyDown And ((Shift And vbShiftMask)>0) Then
        If i <cmb Enumerations. ListCount-1 Then
                Temp=cmb_Enumerations. List(i)
                cmb_Enumerations. RemoveItem i
                cmb_Enumerations. AddItem Temp, i+1
                cmb_Enumerations. ListIndex=i+1
                KeyCode=0
        Else
                Beep
        End If

    End If
    End Sub
```

这个事件处理程序还允许用户重新排列列表项的次序，具体做法是按下 Shift 键的同时使用上下箭头键。请看 Common Value(常用值)信息域，它是表所支持的交互的例子。如果数据类型是枚举型，那么要么是常用值必须出现在枚举列表中，要么 Editable 复选框必须被选中以便用户自己键入常用值。Common Value 域的 LostFocus 事件可确保这样：

```
    Privade Sub txt CommonValue_LostFocus()
    If cmb_DataType. Text=" Enumerated" And txt_CommonValue. Text>"" Then
        If InEnumerations(txt_CommonValue. Text)Then
            Exit Sub
        End If
        If MsgBox("The common value"+txt_CommonValue. Text+_
                "does not appear in your choice"+"of enumerations. Would you like to add
it?",_
                vbYesNo+vbQuestion,"Custom Property")=vbYes Then
            cmb Enumerations. AddItem txt commonValue. Text
            cmd_OK. Enabled=VerifyOk
        Else
            If chk_Editable <> vbChecked Then
                MsgBox " The 'User can type data directly' box on the Access page has been
GetCheck." vbInformation." Custum Property"
                chk_Editable=vbChecked
```

```
            End If
        End If
    End If
    End Sub
```
回到 PropertiesClass，我们看看 AddCustomProperty 方法：
```
    Public Sub AddCustomProperty()
    Dim Property As New CustomPropertyForm
    Property. Show vbModal
    If Property. OK Then
        Property. Selected=True
        Base. Add Property
    Else
        Unload Property
    End If
    End Sub
```

表/类的实例被创建出来，并命名为 Property。它作为一个模式表显示：用户敲击了 OK 或 Cancel 按钮之后，便返回到本过程；OK 属性可以揭示用户按动的是 OK 还是 Cancel。如果按动的是 Cancel 按钮，我们就用不着这个用户属性对象了。但是，还不能任其自生自灭；必须要显式地下载表。

如果按动的是 OK 按钮，我们可以把新的对象加入 List 集合，使用的是我们已经见识过的、ListablesClass 类的内部方法，还有支持对重复项等等问题的处理。从用户属性的角度看来，编辑一个已有的用户属性要简单得多：
```
    Public Sub EditCustomProperty(Cescription As String)
    Base. Edit Description
    End Sub
```

在此之前，我们并没有讨论过 ListablesClass 类的 Edit 方法，原因是我担心如果不结合着它的应用环境来讲解就讲不清楚。您知道，Edit 方法只是在编辑用户属性表(或者我们马上要见到的用户事件表)的时候才被调用，它的代码应该没什么难的：
```
    Public Sub Edit(ByVal Description As String)
    List(Description). Show vbModal
    If Description <> List(Description). Description Then
            Dim Listable As Object
            Listable=List(Description)
            List Remove Description
            Add Listable
            Description=Listable. Description
    End If
    List(Description). Selected=True
```

```
End Sub
```

VBX Genie 完成任务并下载时，应用程序必须下载加入 List 的所有用户属性表(或者用户事件表)。这件事情由 Prime 表的 Unload 事件来做：

```
Private Sub Form_Unload(Cancel As Integer)
Set Properties=Nothing
Set Events=Nothing
While Forms. Count〉1
  Unload Forms(Forms. Count-1)
Wend
End Sub
```

在这个处理程序中，我们显式地清除了 Properties 集合和 Events 集合，只有它们可能还包含有表。在这之后，我们可以检查 Forms 集合以确保没有遗漏。

与其他集合不同，内部 Forms 集合的索引从0开始。此时正下载的 Prome 表肯定是0号元素，所以我们可从1号表元素开始检查。

8.8 支持 Events 页

人们对熟悉的事情效率更高，即所谓的熟能生巧；基于这个事实的考虑，我把 Events 页设计得跟 Properties 相同。图8.7是 Events 页。

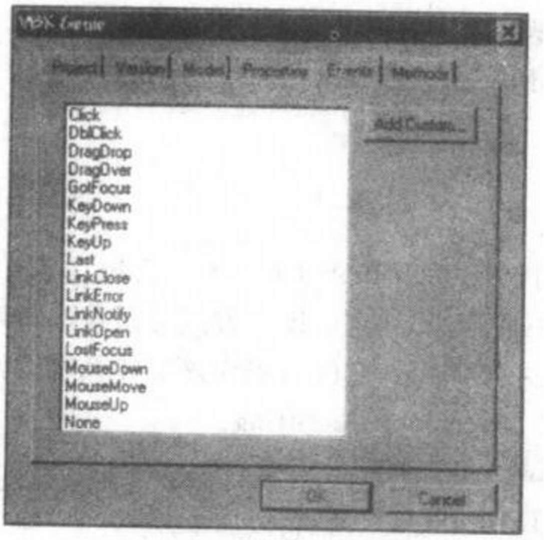

图8.7　VBX Genie 的 Events 页

EventsClass 类也与 PropertiesClass 类相似；不同之处是事件比属性要简单，因为事件与 MODEL 标志之间不存在交互作用。这并不是说什么交互作用都没有了；事件与属性，事件与方法之间还存在着一些交互作用，但同 MODEL 标志相比，它们要好管理得多。没有"StdEventClass"类，EventClass 对象中(几乎)全都是我们熟悉的 ListablesClass 对象。下面是 EventClass 类的(General)(Declarations)部分：

```
Option Explicit
```

```
Public Base As New ListablesCalss
Private Properties As PropertiesClass
Private Methods As MethodsClass
```

作为事件的集合，它必须囊括所有的标准事件。这在类的 Initialize 事件里完成：

```
Private Sub Class_Initialize()
AddStdEvent" PEVENTINFO_STD_CLICK"," Click", 1
AddStdEvent" PEVENTINFO_STD_DBLCLICK"," DblClick", 1
AddStdEvent" PEVENTINFO_STD_DRAGDROP"," DragDrop", 1
AddStdEvent" PEVENTINFO_STD_DRAGOVER"," DragOver", 1
AddStdEvent" PEVENTINFO_STD_GOTFOCUS"," GotFocus", 1
AddStdEvent" PEVENTINFO_STD_KEYDOWN"," KeyDown", 1
AddStdEvent" PEVENTINFO_STD_KEYPRESS"," KeyPress", 1
AddStdEvent" PEVENTINFO_STD_KEYUP"," KeyUp", 1
AddStdEvent" PEVENTINFO_STD_LAST"," Last", 2
AddStdEvent" PEVENTINFO_STD_LINKCLOSE"," LinkClose", 2
AddStdEvent" PEVENTINFO_STD_LINKERROR"," LinkError", 2
AddStdEvent" PEVENTINFO_STD_LINKNOTIFY"," LinkNotify", 2
AddStdEvent" PEVENTINFO_STD_LINKOPEN"," LinkOpen", 2
AddStdEvent" PEVENTINFO_STD_LOSTFOCUS"," LostFocus", 1
AddStdEvent" PEVENTINFO_STD_MOUSEDOWN"," MouseDown", 1
AddStdEvent" PEVENTINFO_STD_MOUSEMOVE"," MouseMove", 1
AddStdEvent" PEVENTINFO_STD_MOUSEUP"," MouseUp", 1
AddStdEvent" PEVENTINFO_STD_NONE"," None", 2
End Sub
```

前面提到过，标准事件可以是 ListableClass 对象，因此有下面的 AddStaEvent 方法：

```
Private Sub AddStdEvent(Identifier As String,Description As String,ByVal Version As Integer)
Dim Item As New ListableClass
Item. Init Identifier, Description, Version
Base. Add Item
End Sub
```

EventClass 类的大多数其他方法(除了我们还未曾讨论的代码编写方法)都与 PropertiesClass 类里的差不多。包括类的 Terminate 方法，它也必须将用户创建的全部用户事件表下载。

8.9 支持用户事件

用户事件的表比用户属性的要简单些，它的选项要少些，见图8.8。

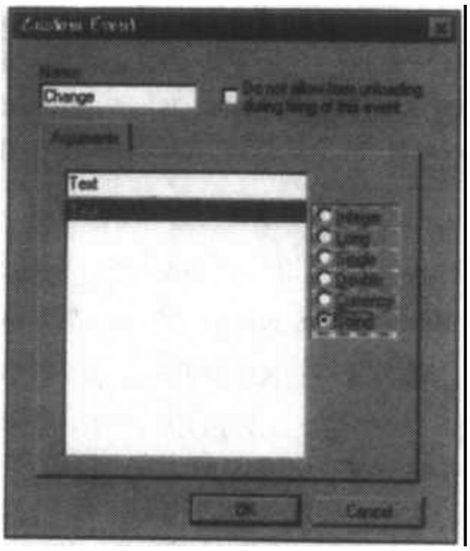

图8.8 VBX Genie 的 Custom Event 对话框

然而，眼见也可能不为实。用户事件的参数可能比事件本身要复杂得多，而大部分代码就耗费在这上面了。

上面是 CustomEventForm 的(General)(Declarations)：

```
Option Explicit

Public OK As Boolean

Public Description As String
Public Selected_As Integer

Private EF_fNoUnload As Boolean
Private Arguments As New Collection

Public UI As ListBox
Public UIIndex As Integer
Private DataTypes As New Collection
```

请注意 CustomEventForm 与 CustomPropertyForm 的区别，在 CustomEventForm 中只有一个标志(EF_fNoUnolad)而不存在标志集合，此外还有 Arguments(参数)集合，其中是与一个用户事件相联系的参数；还有参数的数据类型的集合。

和用户属性一样，用户事件也是在表的 Activate 事件中把数据从属性传给表控件的：

```
Private Sub Form_Activate()
cmd_OK. Enabled=VerifyOK
OK=False
txt_Description. Text=Description
chk_NoUnload=EF_fNoUnload
SetArguments cmb_Arguments
End Sub
```

事件参数以两种方式保存。一种是作为 EventArgClass 对象，我们一会儿讨论；另一种是保存在组合框的 ItemData 数组属性中，与表上的数据类型选项按钮相对应。

过程 SetArguments 把数据从 EventArgClass 对象中一一分离出来：

```
Private Sub SetArguments(c As ComboBox)
c. Clear
Dim i As Integer
For i=1 To Arguments. Count
    c. AddItem Arguments. Item(i). Description
    c. ItemData(c. Newindex)-Arguments. Item(i). DataType. VB_Code
Next i
End Sub
```

在这个过程中，我们遍历整个 Arguments 集合，把每个 Description 字段(即参数名称)加入组合框，并把参数的数据类型复制到该项的 itemData 字段。

当用户单击 OK 按钮，上述过程的逆过程将发生：

```
Private Sub GetArguments(c As ComboBox)
Set Arguments-New Collection
Dim i As Integer
Dim Arg As EventArgClass
For i=1 To c. ListCount
    Set Arg=New EventArgClass
    Arg. Description=c. List(i-1)
    Set Arg. DataType=DataTypes(c. ItemData(i_1))
    Arguments. Add Arg
Next i
End Sub
```

过程中第一条语句不但创建一个新的集合，而且还自动地把旧的集合释放掉！如果要用(可能)变化了的数值重建集合，这倒是一条捷径。(请别忘了，本表将被用来编辑和添加一个用户事件。)新的 EventArgClass 对象就在这里被创建。

每个事件参数都与一个数据类型相联系。在不同的场合，我们不得不以不同的方式对这些数据类型加以解释。此时，它们必须以选中的选项按钮的形式出现。稍后在我们生成 VBX 代

码的时候，得利用它们来产生正确的 VBX CDK 常数和标志。在传统的应用程序里，代码是分散的。在面向对象程序中，分散仍然是必要的。但至少代码要经过打包处理，这样我们编写了代码之后就可以把它抛在脑后。

EventArgDataTypeClass 很简单，它保存一个数据类型的有关信息：

```
Option Explicit

Public Name As String
Public VB_Identifier As String
Public VB_Code As Integer

Public VBX_Identifier As String
```

EventArgDataTypeClass 类的代码就是填充各个字段的函数：

```
Public Sub Init(aName As String,aVB_Identifier As String,ByVal aVB_Code As Integer,_
        aVBX_Identifier As String)
Name=aName
VB_Identifier=aVB_Identifier
VB_Code=aVB_Code
VBX_Identifier=aVBX_Identifier
End Sub
```

CustomEventForm 被创建时，EventArgDataTypeClass 对象也被创建；DataTypes 集合在表的 Initialize 事件处理程序中初始化：

```
Private Sub Form_Initialize()
Dim dtNull As New EventArgDataTypeClass
dtNull. Init" Null"," vbNull", vbNull," -Unused-"
DataTypes. Add dtNull," Null"

Dim dtInteger As New EventArgDataTypeClass
dtInteger. Init" Integer"," vbInteger", vbInteger," ET_12"
DataTypes. Add dtInteger," Integer"

Dim dtLong As New EventArgDataTypeClass
dtLong. Init" Long"," vbLogn", vbLong," ET_14"
DataTypes. Add dtLong," Long"

Dim dtSingle As New EventArgDataTypeClass
dtSingle. Init" Single"," vbSingle", vbSingle," ET_R4"
DataTypes. Add dtSingle," Single"
```

```
Dim dtDouble As New EventArgDataTypeClass
dtDouble. Init" Double"," vbDouble", vbDouble," ET_R8"
DataTypes. Add dtDouble," Double"

Dim dtCurrency As New EventArgDataTypeClass
dtCurrency. Init" Currency"," vbCurrency", vbCurrency," ET_CY"
DataTypes. Add dtCurrency," Currency"

Dim dtDate As New EventArgDataTypeClass
dtDate. Init" Date"," vbDate", vbDate," -Unused-"
DataTypes. Add dtDate," Date"

Dim dtString As New EventArgDataTypeClass
dtString. Init" String"," vbString", vbString," ET_HLSTR"
DataTypes. Add dtString," String"
End Sub
```

集合经过精心安排，使得集合元素的索引正好是数据类型的 Visual Basic 常数。换句话说，vbNull 所代表的数据类型被保存为 DataType(1)，而 vbNull 正好等于1。别的象 vbInteger 等也类似。

用对象表示数据类型的好处，在下章开始生成代码时我们将有深切体会。

另外，尽管我们不断引用这样一些数据类型——比如，把选中的数据类型对象从 DataTypes 集合复制为 EventArgClass 对象——您也不必担心它们在该删除时删除不了，或是在不该删除时却被删除了。每个 Visual Basic 对象都有一个内部的引用计数，仅当对象不再被引用，它才被释放。

8.10 加入 Methods 页

解决了属性和事件，就只剩下方法了。图8.9显示的是 Methods 页。

您知道，VBX 没有“用户方法”一说；因此在页上不会有“Add Custom…(加入用户……)”按钮。除此之外，本页与前面见到的那些都类似。MethodsClass 类甚至比 EventsCalss 还简单。它的(General)(Declarations)如下：

```
Option Explicit

Private Base As New ListablesClass
Private Events As EventsClass
Public UI As ListBox
```

就象 EventsClass，MethodsClass 也能够直接使用 ListableClass 对象。各个对象在 Initialize

图8.9 VBX Genie 的 Method 页

过程里安排妥当：

```
Private Sub Class_Initialize()
AddStdMethod" METH_ADDITEM"," AddItem", 1
AddStdMethod" METH_CLEAR"," Clear", 2
AddStdMethod" METH_DRAG"," Drag", 1
AddStdMethod" METH_LINKSEND"," LinkSend", 2
AddStdMethod" METH_MOVE"," Move", 1
AddStdMethod" METH_REMOVEITEM"," RemoveItem", 1
AddStdMethod" METH_REFRESH"," Refresh", 1
AddStdMethod" METH_ZORDER"," ZOrder", 2
End Sub
```

AddStdMethod 与 AddStdEvent 太相似了，我不准备在这里多讲。类中的其他方法大多数也与我们所见到的大同小异。Selected 属性函数在原理上同其他类相似，但它的实现则不一样；它反映出方法和事件之间的相互依赖关系：

```
Public Property Let Selected(ID As Variant,Value As Boolean)
Static Busy As Boolean
If Busy Then Exit Property
Busy=True
If ID=" Drag" Then
        Events. Selected("Drag")=Value
        Base. Item("Drag"). Selected=Value
ElseIfLeft $ (ID,4)-" Link" Then
        If Value Then Events. Selected("Link")=Value
        Base. Item("LinkSend"). Selected=Value
ElseIfID=" AddItem" Or ID=" RemoveItem" Then
        Base. Item("AddItem"). Selected=Value
```

```
        Base. Item("RemoveItem"). Selected=Value
        If Value Then
                Base. Item("Clear"). Selected=True
        End If
Else
        Base. Item(ID). Selected=Value
End If
Busy=False
End Property
```

爱刨根问底的您也许会问："那么，方法和属性之间的相互依赖关系又是怎样的呢?"其实，由于存在着"扩散"效应，有方法和事件相关联就已经足够。以用户选中 LinkSend 方法为例。如前所示，这个选中的动作将使所有 Link 事件都被选中。如果您参阅前面的 EventsClass. Selected 方法，您会发现它保证选中或不选中所有与 Link 有关的属性，这同样适用于与 Link 有关的事件。

那么，用户选中与 Link 有关的属性时，是否同时也选中了 LinkSend 方法呢?不是，因为是否处理 LinkSend 方法消息，完全是可选的；如果 VBX 不显式地处理这些消息，VBDefControlProc()将隐式地处理它们。

第九章 生成代码

VBX Genie 的最终目标当然是为了生成代码。然而，这反倒是整个工作中较为简单的一部分；这种情况在通常的 Windows 应用程序中也经常见到，在程序中占用绝大多数代码行的恰恰是用户界面。

尽管如此，代码生成的工作还是很有意思的。而且，我在 VBX Genie 中用到的技巧是我在别处不曾见到过的，尽管对我来说这些技巧是很平常很自然的。

我们首先还是从研究“资源”这个 Visual Basic 4.0的新特色来开始我们的航程吧！

9.1 加入代码脚本(Code script)

C 和 C++的 Windows 程序员大概都知道“资源池(reource pool)”：即终端用户所见的那些对象的集合，包括图标、信息字符串、菜单和对话框。使用资源池，部分原因是便于国际化，同时这确实又是个良好的设计思路，所有这些东西都与应用程序的代码分离。在不必触动应用程序的情况下它们就能够被修改；实际上，如果使用先进的编辑工具，替换掉它们后甚至根本不必重新编译应用程序！

但是对于 Visual Basic 程序员来说，一直到 VB3才能享受这个便利。要生成最终的 VBX 代码，VBX Genie 必须访问代码脚本：VBX Genie 的第一个版本不得不把这些脚本保存在一个单独的文件里。这样就得处理文件丢失的可能情形。而且，除了脚本资源，我们还得留意工具箱位图和图标：它们也得一个不差地复制到目标目录中。而终端用户却是粗心得让人放心不下。

好啦，问题可以解决了。尽管 VB4不提供处理资源的工具，但是它确确实实地支持资源这一概念。这样就行了；想要编写 VBX 的话，您可以选用 Microsoft 公司的 App Studio(Microsoft Visual C++带有)或 Borland 公司的 Resource Workshop(Borland C++带有)。它们都能够把“资源描述文件”编译为二进制的.RES 文件，再由 Visual Bassic 把.RES 合并到最终的.EXE 文件中。

如今在使用 App Studio 或 Resource Workshop 的时候，我们大概已经不记得过去只能用文本编辑器、完全以资源描述文件的方式创建资源的日子了。然而，新工具只能对标准资源提供直接支持，如菜单、对话框等等。而资源描述文件除标准资源之外，还支持用户自定义资源。那就是说，您可以“杜撰”自己的资源类型名字！如果您想要一个“Flashlight”(手电)或“Sandbar”(沙洲)资源，也没有什么不可以。目前，我需要一些新的资源类型，如“Module”(模块)和“Toolbox”(工具箱)。在我把新资源类型的脚本做好之后，就只需在 VBXGEN.RC 里引用它们即可：

```
8000    TOOLBOX8000.bmp
8001    TOOLBOX8001.bmp
8003    TOOLBOX8003.bmp
8006    TOOLBOX8006.bmp
```

```
MAIN      TOOLICON main. ico

DEF     MODULE main. def

HEADER     MODULE internal. h
MAIN      MODULE main. c
HELP      MODULE help. c
RC MODULE main. rc
VISUAL MODULE visual. c
MAK     MODULE main. mak

PORTAGDI_H     MODULE portagdi. h
PORTAGDI_CPP MODULE portagdi. cpp
VBXXX_H   MODULE vbxxx. h
VBXXX_CPP      MODULE vbxxx. cpp
MAIN_CPP     MODULE main. cpp
```

就象某些标准的资源那样，用户自定义资源也是以一个名字开头，接着是资源类型(您可以自己编造)，接下来是包含有资源的文件名。

如果启动 App Studio 或 Resource Workshop，打开这个. RC 文件，再用 File…Save As 命令把它保存为. RES 文件，那么您在 RC 文件里指定的那些文件将被逐字节地复制成一个可检索的、组织好的形式。由此得到的 VBXGEN. RES 文件可添加到 VBX Genie 工程中。

如果使用的是 App Studio，请务必使用 Visual C++ 1.5x 的16位版本。32位版本产生的是32位的资源，它与 VBX 的16位模块不兼容。

要把文件加入到工程，最简便的办法是把它们从文件管理器或 Windows 资源管理器(依 Windows 的版本不同而有区别)拖动进去。图9.1是加入资源池后的 Project 窗口。

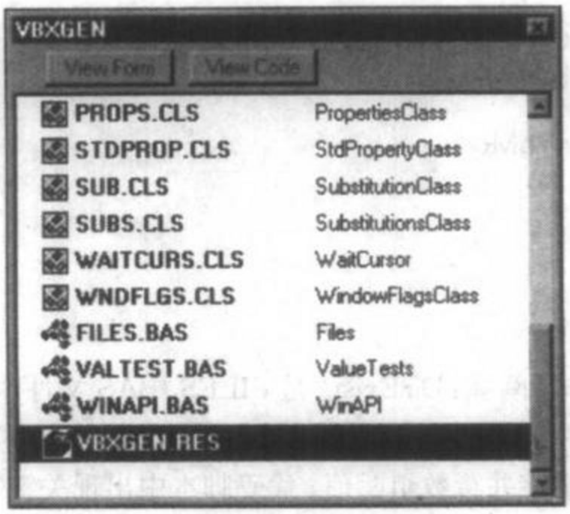

图9.1 加入了资源池的 Project 窗口

9.1.1 创建代码脚本

代码脚本与本书早先描述的 VBX 框架和 VBX++文件基本上是一样的。这里不准备罗列代码脚本的所有内容；否则将占用很大篇幅！

本质上，脚本是 C 或 C++语言的代码模块，其中有一些称为替换体的特殊命令。这些命令以“$”开头，均为大写字母，这样它们看起来更醒目一些。在替换体出现的地方，VBX Genie 将填入一些生成的代码。脚本其余部分原样复制。

9.1.2 生成代码

代码生成工作其实可分为两部分。第一部分是创建替换值，这看起来容易，其实也有一定难度。第二部分是实际地“装配”出文件，它并不如您想象的那么难。

所有这些当然都是在用户单击 Prime 表上的 OK 按钮之后才开始的：

```
Private Sub cmd_OK_Chck()
Dim WC As WaitCursor
Set WC=New WaitCursor
Dim CodeGen As New CodeGenerator
If DirExists(txt_Directory. Text)Then
    CodeGen. Directory=txt_Directory. Text
    CodeGen. Substitute(" $PROJECT")=txt_Name. Text
    CodeGen. Substitute(" $DIRECTORY")=txt_Directory. Text
    DefineVersion CodeGen
    DefineModel CodeGen
    DefineProperties CodeGen
    DefineEvents CodeGen
    DefineMethods CodeGen
    If opt_C. Value Then
        CodeGen. TargetLanguage=" C"
    Else
        CodeGen. TargetLanguage=" C++"
    EndIf
    CodeGen. Show vbModal
    Unload Me
End If
End Sub
```

WaitCursor 我已经作过阐释：DirExists 是 FILES. BAS 文件中的一个工具函数。CodeGeneralor 又是一个类——跟用户属性一样，它也被包装为表。在这个类中包含有一个名为 Substitute 的数组属性。指定关键字并给数组赋值；代码脚本中出现关键字的地方都将被这个值替换掉。

Define Version、DefineModel 和其他类似的调用，其作用都只是为了把这个子过程分割成

更小的模块，而不至于变成一个庞然大物。

DefineVersion 读取表上与 VBX 版本有关的各个控件的值，并据此完成替换：

```
Private Sub Define Version(CodeGen As CodeGenerator)
CodeGen. Substitute(" $FILEVERSIONTEXT")=txt_FileVersion(0). Text+"," +_
        txt_FileVersion(1). Text+"," +txt_FileVersion(2). Text+"," +txt_FileVer-
sion(3). Text
CodeGen. Substitute(" $FILEVERSION")=txt_FileVersion(0). Text+"," +_
        txt_FileVersion(1). Text+"," +txt_FileVersion(2). Text+"," +txt_FileVer-
sion(3). Text
CodeGen. Substitute(" $PRODUCTVERSIONTEXT")=txt_ProductVersion(0). Text
+"," +_
        txt_ProductVersion(1). Text+"," +txt_ProductVersion(2). Text+"," +_
        txt_ProductVersion(3). Text
CodeGen. Substitute(" $PRODUCTVERSION")=txt_ProductVersion(0). Text+","
+_
        txt_ProductVersion(1). Text+"," +txt_ProductVersion(2). Text+"," +_
        txt_ProductVersion(3). Text
CodeGen. Substitute(" $COMPANYNAME")=txt_CompanyName. Text
CodeGen. Substitute(" $PRODUCT")=txt_ProductName. Text
CodeGen. Substitute(" $AUTHOR")=txt_Author. Text
CodeGen. Substitute(" $RELEASEYEAR")=txt_ReleaseYear. Text
CodeGen. Substitute(" $COPYRIGHT")=txt_Copyright. Text
End Sub
```

DefineModel 不那么好懂：

```
Private Sub DefineModel(CodeGen As CodeGenerator)
Select Case Val(cmb_VBVersion. Text)
        Case 1
                CodeGen. Substitute(" $VBVERSION")=" VB100_VERSION"
        Case 2
                CodeGen. Substitute(" $VBVERSION")=" VB200_VERSION"
        Case 3
                CodeGen. Substitute(" $VBVERSION")=" VB200_VERSION"
End Select
CodeGen. Substitute(" $MODELFLAGS")=ModelFlags. Code
CodeGen. Substitute(" $CLASSFLAGS")=ClassFlags. Code
CodeGen. Substitute(" $WINDOWFLAGS")=WindowFlags. Code
CodeGen. Substitute(" $DEFCONTROLNAME")=txt_DefControlName. Text
CodeGen. Substitute(" $CLASSNAME_CPP")=LCase $ (txt_ClassName. Text)
CodeGen. Substitute(" $CLASSNAME")=txt_ClassName. Text
```

```
If cmb_ParentClass. Text=" (None)" Then
        CodeGen. Substitute(" $PARENTCLASS")=" NULL"
Else
        CodeGen. Substitute(" $PARENTCLASS")=Quote(cmb ParentClass. Text)
End If
End Sub
```

ModelFlags. Code?ClassFlags. Code?这些玩艺儿是从哪里钻出来的？

在上一章我没有讲它们，因为我想把所有编写代码的部件集中讲解。我的想法是，要让VBX Genie的各个对象都知道如何为自己编写代码。这样的话，在Prime表中，我们可以只关注替换体的名字；而在数据类中，可以把精力集中在代码的生成上。

对于ModelFlagsClass、ClassFlagsClass和WindowFlagsClass，它们各自的Code方法都是一样的：

```
Public Function Code() As String
Code=Flags. CodeFlags
End Function
```

没错，它只是低层ListablesClass. CodeFlags方法的低水平重复：

```
Public Function CodeFlags() As String
Dim Result As String
Dim i As Integer
For i=1 To List. Count
        If List(i). Selected Then
                If Result>"" Then Result=Result+" |"
                Result=Result+List(i). Identifier
        End If
Next i
If Result="" Then Result=" 0"
CodeFlags=Result
End Function
```

我的思路是，取出各个标志，并对集合里所有选中标志进行逻辑OR运算。如果没有一个标志被选中，返回" 0"。因此在任何要求C风格标志集的场合，Result字符串都能胜任 。

我最初尝试编写的代码生成器，其复杂程度让人简直不堪忍受。我是把脚本分割为小段，把大部分工作压在脚本上。有了这个教训，我决心要把DefineProperties编写得简单些。它同时输出C和C++的代码；由于有些替换体是共享的，完成一次所花费的时间很少，所以没有理由不按下面的做法行事：

```
Private Sub DefineProperties(CodeGen As CodeGenerator)
CodeGen. Substitute(" $VBDATALIST")=Properties. CodeVbDataList
CodeGen. Substitute(" $PROPERTYLIST_CPP")=Properties. CodePropertyListCPP
CodeGen. Substitute(" $PROPERTY_INITIALIZATIONS_CPP")=
```

```
    Properties. CodePropertyInitsCPP
CodeGen . Substitute ( " $ PROPERTYAPPENDS_ CPP " ) = Properties . CodeProp -
ertyAppendsCPP
CodeGen. Substitute(" $PROPERTYLIST")=Properties. CodePropertyList
CodeGen. Substitute(" $ONSETPROPERTY")=Properties. CodePropertySet
CodeGen. Substitute(" $ONGETPROPERTY")=Properties. CodePropertyGet
CodeGen. Substitute(" $ONLOADPROPERTY")=Properties. CodePropertyLoad
CodeGen. Substitute(" $ONSAVEPROPERTY")=Properties. CodePropertySave
CodeGen. Substitute(" $ONCHECKPROPERTY")=Properties. CodePropertyCheck
CodeGen . Substitute ( " $ ONGETSTRINGPROPERTY " ) = Properties . CodeProper -
tyGetstring
End Sub
```

我们仅以一个编写代码方法作为所有编写代码方法的代表，对它加以说明。下面是 CodeVbDataList，它生成的是 VISUAL. C 中 VbData 结构的元素。

```
Public Function CodeVbDataList() As String
Dim i As Integer'
Dim Result As String
For i=1 To Base. Count
    If Base. Item(i). Selected and Not Base. Item(i). Standard Then
            If Base Item(i). Direct StoreThen
        If Result <>"" Then
            Result=Result+vbCrLf
            End If
            Result=Result+vbTab+Base. Item(i). CodeStorageType+"" +_
                Base. Item(i). Description+";"
        End If
    End If
Next i
If Result="" Then Result=vbTab+" UINT: 16;"
CodeVbDataList=Result
End Function
```

可见，PropertiesClass 对象检查其集合内的每一个属性、选出那些被选中的、非标准(换句话说，是用户)的属性。然后它检查它们是否直接由 VBX 管理器存储。

```
Public Function DirectStore() As Boolean
DirectStore=ModelFlags. Base("PF_fGetData"). Selected Or_
ModelFlags. Base("PF_fLoadMsgOnly"). Selected Or_
ModelFlags. Base("PF_fSaveData"). Selected Or_
ModelFlags. Base("PF_fSetData"). Selected
```

```
End Function
```

如果选中的用户属性是直接由 VBX 管理器存储，那么将得到属性的数据类型和名称，以制表符和换行符相隔，这样不但看着顺眼，而且能够保证在目标结构中正确编译。

所有编写代码的方法看起来都大同小异。各个编写代码的方法都是要在目标脚本的某个位置生成代码；各个编写代码的方法都与特定的替换名联系在一起。这一切都是在 CodeGenerator 表处于图9.2所示阶段时发生的。

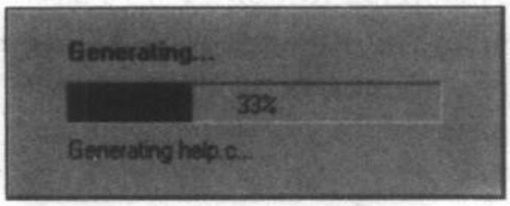

图9.2 CodeGenerator 表

表与类一样，也包括实现其各项功能的属性：

```
Option Explicit

Private Substitutions As New SubstitutionsClass

Private Steps As Integer
Privatc CurrcntStep As Integer

Private TargetIDE As String
Public Language As String

Private VBAPI Path As String
```

SubstitutionsClass 是 SubstitutionClass 对象的集合；CodeGenerator 的 Subtitute 方法允许它加入集合元素：

```
Public Property Let Substitute(aGeneric As String,aValue As Variant)
Substitutions Substitute(aGeneric)=aValue
End Property
```

表的 Active 事件处理程序具体生成输出文件：

```
Private Sub Form_Activate()
Dim W As WaitCursor
Set W=New WaitCursor
Select Case Language
Case" C"
    GenerateC
Case" C++"
```

```
        GenerateCPP
    End Select
    Dim Message As String
    Message=" You VBX Project has been generated"
    Select Case TargetIDE
    Case" MSVC150"
        If AddToProject_MSVC150 Then
            Message=Message+" and loaded"
        End If
    Case" BC45"
        If AddToProject_BC45 Then
            Message=Message+" and loaded"
    End If
    End Select
    MsgBox Message+".", vbinformation
    Unload Me
    End Sub
```

显然是 GenerateC 和 GernerateCPP 在实实在在地做着生成代码的工作。我们来看看 GenerateC:

```
    Private Sub GenerateC()
    If TargetIDE=" MSVC150" Then
            Steps=13
            GenerateTextFile" MAK"," MODULE", Prime. txt_Name+". mak", True
    Else
            Steps=12
    End If
    GenerateTextFile" DEF"," MODULE"," main. def", True
    GenerateTextFile" HEADER"," MODULE"," internal, h. ", False
    GenerateTextFile" MAIN"," MODULE"," main. e", True
    GenerateTextFile" HELP"," MODULE"," help. c", False
    GenerateTextFile" RC"," MODULE"," main. rc", True
    GenerateTextFile" VISUAL"," MODULE"," visual. c", True
    GenerateBinaryFile 8000," TOOLBOX"," 8000. bmp"
    GenerateBinaryFile 8001," TOOLBOX"," 8001. bmp"
    GenerateBinaryFile 8003," TOOLBOX"," 8003. bmp"
    GenerateBinaryFile 8006," TOOLBOX"," 8006. bmp"
    GenerateBinaryFile 8006," TOOLBOX"," 8006. bmp"
    GenerateBinaryFile " MAN"," TOOLICON"," main. ico"
    End Sub
```

Steps 是一个模块变量，它用来计算进度条(progress bar)的填允量。

文件是GenerateBinaryFile 和GenerateTextFile 生成。这两个函数中的第一个函数要简单些：

```
Private Sub GenerateBinaryFile(Index As Variant,_
          Format As String,FileName As String)
UpdateUI FileName
Dim Buffer As String
Buffer=LoadResData(Index,Format)
On Error Resume Next
Kill FileName
Open FileName For Binary Access Write As #1
Put#1, 1, Buffer
Close#1
DoEvents
End Sub
```

LoadResData 是 Visual Basic 新提供的函数；给它一个资源的名称(如“index”)和类型(如“MODULE”)，它就在缓冲区填入内容。需要注意一点：资源在装入时作了填充处理，因此实际字节数可能比您预料的要多几个字节。对二进制资源来说这不算是问题，因为它们的结构通常也被填充至同样的边界；但是，在遇到文本资源时，应考虑到这点，要把文本分割为单独的行、然后每次一行地输出：

```
Private Sub GenerateTextFile(Index As Variant,_
          Format As String,FileName As String,_
          ByVal Template As Boolean)
UpdateUI FileName
Dim Lines As New Collection
LoadResText Index, Format, Lines
IfTemplate Then Transform Lines
On Error Resume Next
Kill FileName
Open FileName For Output As#1
Dim i As Integer
For i-1 To Lines. Count
          Print#1, Lines(i)
Next i
Close #1
DoEvents
End Sub
```

多数文件可以按原样复制；有的文件包含有替换体，故必须要经过“变换”。这样的文件

称为模板(template)，函数中 Template 参数就是说明当前文件是否为模板。如果是模板，就调用 Transform 过程：

```
Private Sub Transform(Lines As Collection)
Dim i As Integer
Dim Command As String, Arg As String
i=1
While i <=Lines. Count
        Command=GetCommand(Lines(i), Arg)
        IfLeft $ (Command,3)=" $IF" Then
                If(Command=" $IFDEF"And Len(Arg)=0) Or
                                (Command=" $IFNDEF" And Len(Arg)) 0) Then
                        Do
                                Lines. Remove i
                                Command=GetCommand(Lines(i), Arg)
                        LooP Until Command=" $ENDIF"
                Else
                        Lines. Remove i
                End If
        ElseIf Command=" $ENDIF" Then
                Lines. Remove i
        ElseIf InStr(Lines(i):" $")) 0 Then
                Dim Temp As String
                Temp=Substitutions. Transform(Lines(i))
                IfTemp <> Lines(i) Then
                        Lines. Remove i
                        Lines. Add Temp,, i
                End If
                i=i+1
        Else
                i=i+1
        End If
Wend
End Sub
```

这只是标准的字符串操作：如果研究一下输入的脚本和这里的代码，您就不难理解。Substitutions 集合就是在这里引用的；它遍历所包含的每个 SubstitutionClass 对象，并在需要时进行实际的替换：

```
Public Function Transform(ByVal Text As String)As String
Dim i As integer
For i=1 To List. Count
```

```
        If InStr(Tex," $ ")=0 Then Exit For
        Text=List(i). Transform(Text)
Next i
Transform=Text
End Function
```

底层的 Transform 完成实际的字符串操作：

```
Public Function Transform(Text As String)As String
Dim i As Integer
i=InStr(Text,Generic)
While i>0
        Text=Left $ (Text,i-1)+Value+Mid $ (Text,i+Len(Generic))
        i=InStr(Text,Generic)
Wend
Transform=Text
End Function
```

这些相当精焊、而且不太复杂的代码片断组合到一起，结果导致一打左右的文件的生成，用户新的 VBX 工程由此产生。

根据用户运行 VBX Genie 的环境，工程将自动地由 Microsoft Visual C++或 Borland C++打开。同这些 IDE 通信的代码(相当冗长！)将在下章加以阐述。

第十章　自安装 IDE 工具

在前面几章开发完成的 VBX Genie，可以生成 C 或 C ++语言代码，代码再编译成为 VBX。这当然要用到编译器。因而，尽管 VBX Genie 可以单独运行，但把它作为工具并与 C/C ++集成开发环境(IDE)(如 Microsoft 公司的 App Wizard 或 Borland 公司的 App Expert 等等)相结合，这样会更好一些。

这样的工具应能访问 IDE 的有关信息，比如当前的工作目录。而且它不但要能够生成 VBX 代码，还要能够生成 VBX 工程。

在本章里，我们来学习怎样把 VBX Genie 加到两个最流行的开发环境的工具菜单中，以及怎样把您编写的任何工具加入工具菜单中。

10.1　把 VBX Genie 加入开发环境

您可以于工地把 VBX Genie 加到 Microsoft Visual C++1.52 的工具菜单中。那并不难，只需用 Option... Tools 命令并在对话框中填写一些信息域。但这样的话，在运行 VBX Genie 之后，您仍得创建一个 MSVC 工程并把 VBX Genie 生成的文件加入其中。您得指定合适的工程类型，要记住关闭“Use Microsoft Foundation Classes”选项，并把 VBAPI.LIB 加到工程中。

看这有多烦琐！VBX Genie 本该把所有这一切都替您做了。Microsoft 公司的 Control Wizard(用于生成 OCX)就是这样。Microsoft 能做的任何事情，我们都可以做得更好，至少只需花费更少的钱。

我们应支持哪种 IDE？Microsoft Visual C ++2.0 和 4.0 不在考虑之列；因为不可能编写出 32 位的 VBX，而 MSVC2.0 和 4.0 只能产生 32 位机器代码。我们希望能够支持 MSVC1.5x 及 Borland C ++4.x。也许除此之外，您还想要支持其他一些 IDE；可我帮不了您，我没有它们的拷贝。

这件工作包括三个方面：

1. 我们需要加入代码，以便首次运行 VBX Genie 时完成自动安装。从中我们还可以得到其他好处。如果处处留意隔离代码的问题，那么我们的“InstallTool(安装工具)”表可以重用在其他类似工具的安装上。

2. 我们要为工程生成适当的 .MAK 或 .IDE 文件。

3. 为了打开新创建的工程，需要发送击键给 MSVC 或 BCW，因此我们还要加入这方面的代码。

10.2　给我一个支点

VBX Genie 需要知道它是为 MSVC1.5x 还是为 BCW4.x 创建工程，这一要求并不过分。但它如何得知呢？

这简单：我们让 IDE 在命令行标榜自己。例如：

VBXGEN/IDE=MSVC150

这样做还有一个额外的好处：略去命令行的这个标志，我们知道 VBX Genie 不是从 IDE 运行的，因而可以认为运行它是为了安装自己。

为了检测出这一情况，我们在 Prime 表的 Load 事件处理程序中添加一些代码：

```
Private Sub Form_Load()
IniDatal.filename=MakePath("",App.Path,App.EXEName,".ini")
If Not CheckInstallation Then
        Unload Me
        Exit Sub
End If
    〈awr〉
    〈awr〉
```

CheckInstall 判断 VBX Genie 是否是单独运行；如果是这样，它将装入并初始化一个新的工具安装表，由这个表负责安装，然后返回：

```
Private Function CheckInstallation() As Boolean
If InStr(UCase$(Command$)," /IDE=") =0 Then
    Load InstallTool
    InstallTool.Imagel.Picture=Icon
    InstallTool.MSVC200.Enabled=False
    InstallTool.Show vbModal
    CheckInstallation=False
    End
Else
    CheckInstallation=True
End If
End Function
```

InstallTool 表是如何工作的呢?还有，为什么最开始要禁止一个控件?我们先来看看它的外观。图10.1是 Installing VBX Genie 对话框。

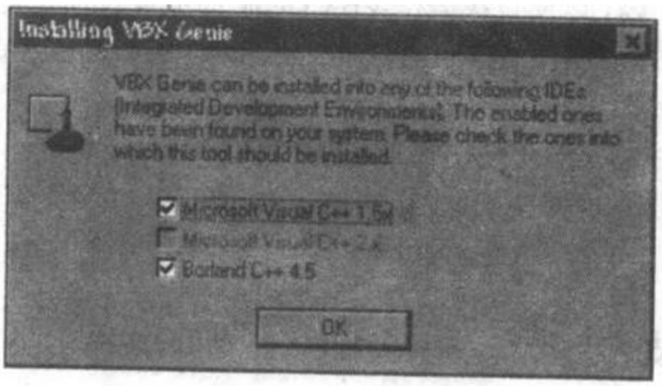

图10.1　Installing VBX Genie 对话框及选项

注意 "Microsoft Visual C ++2.x" 复选框是失效的。我前面提到过，不能用 MSVC2.x 创建 VBX。在本表中包括 Microsoft Visual C ++2.x 这一项，只是考虑到表的可重用性和让用户能够熟悉，就如同 Open File 对话框那样。尽管在标题栏和正文中引用了 "VBX Genie"，但只要稍加改动，本表就可用于其他工具的安装。

设计时的 InstallTool 看起来有细微的差别(见图10.2)，这是由于它的有待用户化的正文和图象控件，以及在运行时不可见的控件所致。

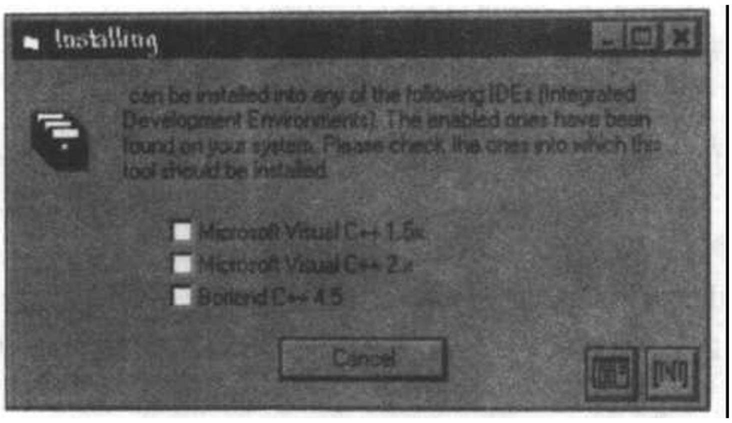

图10.2 安装对话框的选项选择

除了 IniData 控件(InstallTool 有自己的 IniData 控件)，还有标题、正文和图标都是不相同的。

InstallTool 表第一次被装入时，它修改自己的标题和正文：

```
Private Sub Form_Load()
Caption=Caption+App. Title
Labell=App. Title+Labell
End Sub
```

表的 Caption 和 Labell(文本块)加入了从 App. Title 属性得到的应用程序的标题。(这个值来源于 Make EXE File 对话框。)因此无需多费气力，至少表的面貌已是可重用的了。

装入表后，令 MSVC2. x 的复选框失效：

```
InstallTool. MSVC200. Enabled=False
```

InstallTool 表自身并不拘泥于这个思路。表上的三个复选框最初都是激活的；而 InstallTool 知道您可能要排除其中某个。所以它等到 Load 事件结束之后，直到 Active 事件(此时调用者真正地显示出表)时，再来检查三个工具的状态。定位例程的任务是，如果在用户系统中某个工具环境缺失，那么令此工具环境对应的复选框失效，而仍然有效的那些复选框则被选中：

```
Private Sub Form_Activate()
LocateAvailableTools
MSVC150. Value=-MSVC150. Enabled
MSVC200. Value=-MSVC200. Enabled
BC45. Value=-BC45. Enabled
End Sub
```

LocateAvailableTools 负责做那些杂务，但它只寻找仍处于激活状态的那些复选框所对应的工具环境，看这工具环境是否可用：

```
Private Sub LocateAvailableTools()
```

```
MSVC150.Enabled=IsMSVC150_Available(MSVC150.Enabled)
MSVC200.Enabled=IsMSVC200_Available(MSVC200.Enabled)
BC45.Enabled=IsBC45_Available(BC45.Enabled)
End Sub
```

这真象是在进行一场大清扫。不管使用的是何种语言，把过程函数编写得简短而高聚焦(focused)都会有一个好处：可减少出错机会。在这里可能需要对三个 IDE 进行定位，因此如果我们把它们分开处理，遵循上面的箴言，那么事情会好办得多。

我们如何知道用户的系统安装了 Microsott Visual C ++ 1.5x 呢?谢天谢地，这不算太难。MSVC1.5x 把它的选项设置保存在初始化文件 MSVC.INI 中。MSVC1.0版本也是；其中有一项告诉我们安装的是哪个版本。这一项为：

```
[Microsoft Visual C ++]
Version=1.50.0000.0000
  〈awr〉
  〈awr〉
```

即使版本为1.52，该项仍然记为1.50；但这不要紧。我们只需读出开头三个字符就万事大吉：

```
Private Function IsMSVC150_Availabie(ByVai ShouldTest As Bootean)As Boolean
IsMSVC150_Available=False
If ShouldTest Then
        IniDatal.FileName=" msvc.ini"
        IniDatal.Section=" Microsoft Visual C ++"
        IniDatal.KeyName=" Version"
        IsMSVC150_Available=_
                (Left $ (IniDatal.Text,3)=" 1.5")
End If
End Function
```

其实我们只检测ShouldTest参数是否为真即可，该参数实际上是复选框的Enabled属性。

查找 Borland C ++的方法与之类似。BCW 也是把一些信息放在初始化文件当中。确实，版本信息并不在其中；然而，从4.0版到4.5x 版，安装工具的方法却都是一样的；所以我随便选了一项来查找，只要保证它在更老的版本(如3.1)里没有就行：

```
Private Function IsBC45_Available_
        (ByVai ShouldTest As Boolean)As Boolean
IsBC45_Available=False
If ShouldTest Then
        IniDatal.FileName=" bcw.ini"
        IniDatal.Section=" IDE"
        IniDatal.KeyName=" DefaultDesktopDir"
```

```
        IsBC45_Available=(IniDatal.Text>"")
    End If
End Function
```

(插一句，如果您还用着 BC4.0，赶紧把它扔掉了事！这个版本有严重错误。4.02版也好不到哪里去。您需要至少4.5版。)

就剩下查找 MSVC2.x 的例程没有讨论了。因为我们在这个工程中用不着考虑 MSVC2.x，所以暂时不用管它。(如果想要在别的工程中使用 InstallTool 表，这一点可以继续考虑。)

Microsoft 公司曾声称，要淘汰.INI 文件，而把这类信息以及 OLE 的有关信息保存在注册数据库(Registration Database)中。他们的想法是，GetPrivateProfileString()等等的一系列函数突然间就由操纵初始化文件转而变为操纵注册数据库了。

但不管什么原因，只要考虑到16位应用程序，那一天就还不会到来，至少在 Windows NT 或 Windows95目前的版本中是这样。用那样一些函数操纵注册数据库复杂程度显然太高。

在 InstallTool 表上没有设置 Cancel 按钮，因为取消复选框并单击 OK 按钮，其效果与 Cancel 是一样的。OK 按钮通过查看每个复选框得知用户想要把 VBX Genie 安装到哪些 IDE 中；然后开始安装：

```
Private Sub cmd_OK_Click()
If MSVC150.Value Or BC45.Value Then
        Locate_VBAPI
End If
If MSVC150.Value Then
        Install_MSVC150
End If
If MSVC200.Value Then
        Install_MSVC200
End If
If BC45.Value Then
    Install_BC45
End If
Hide
End Sub
```

(复选框的 Value 属性永远都不可能真正为“真”(在 VB 中用1表示真值)，但是在作条件检查时它返回“真”。)

第一个函数 Locate_VBAPI 允许用户指定 Visual Basic CDK 构件的位置。作为安装工作的一部分，我们将把这个目录加入到 IDE 对#INCLUDE 头文件和库文件的搜索路径中。这个子例程本身并不复杂，它用到了隐身于表上的公用对话框控件：

```
Private Sub Locate_VBAPI()
CmDialogl.DefauliExt=" lib"
```

```
CmDialogl.DialogTitle=" Locate VBAPI.LIB"
CmDialogl.FileName=" vbapi.lib"
CmDialogl.Filter=" VBAPI.LIB|vbapi.lib|All files|**"
CmDialogl.FilterIndex=1
CmDialogl.Fiags=cdiOFNHideReadOnly Or cdiOFNFileMustExist

On Error Resume Next
CmDialogl.ShowOpen
If Err=0 Then
        VBAPI_Path=Left$(CmDialogl.FileName,_
                Len(CmDialogl.FileName)-10)
        IfLen(VBAPI_Path)=2 Then
                VBAPI_Path=VBAPI_Path+" \"
        End If
        IniDatal.FileName=_
                MakePath("",App.Path,App.EXEName,".ini")
        IniDatal.Section=" VBAPI"
        IniDatal.KeyName=" Path"
        IniDatal.Text=VBAPI_Path
End If
End Sub
```

尽管公用对话框询问的是 VBAPI.LIB 的位置，但 Visual Basic 把头文件 VBAPI.H 也保存在同一目录下；因此找到一个的位置也就找到了全部。然后从文件名称中剥去文件名(以及结束的反斜杠)并存入静态字符串变量 VBAPI_Path 中，该变量也被保存在 VBX Genie 的初始化文件.INI 中。

10.3 把 VBX Genie 安装到 MSVC1.5x

前面提到过，MSVC1.5x 把它的工具配置情况保存在它的.INI 文件中。其中有关键字“ToolNumber”，它的值表示当前所配置工具的数量。而每门工具都有刻画自己的几个关键字名，它们都以工具序号结尾。例如，如果定义有3个工具，那么就可能有这样3个关键字名：“ToolMenuText1”、“ToolMenuText2”、“ToolMenuText3”。在一个不能支持数组的环境里，它被组织成数组的形式。

这一信息相当重要，它可防止同一工具被多次安装。所以安装的第一步是检查以前是否安装过：

```
Private Function GetToolNumber_MSVC150
        (ByVal ToolName As String) As String
Dim ToolCount As Integer, t As Integer
IniDatal.KeyName=" ToolNumber"
ToolCount=Val(IniDatal.Text)
```

```
For t=1 To ToolCount
        IniDatal. KeyName=" ToolMenuText" +Format $ (t)
        If IniDatal. Text=ToolName Then
                GetToolNumber_MSVC150=Format $ (t)
                Exit Function
        End If
Next t

Dim Digits As String
Digits=Format $ (ToolCount+1)
IniDatal. KeyName=" ToolNumber"
IniDatal. Text=Digits
GetToolNumber_MSVC150=Digits
End Function
```

如果工具已安装过，本函数将返回其序号字符串，否则返回其下一个工具的序号(即末位工具序号+1)。同时它还更新 MSVC. INI 文件的 ToolNumber 项。

有了 ToolNumber，就可构造出关键字名，并相应地设定其值。记住在.INI 文件中，如果关键字已经存在，那么它的内容将被更新；如果关键字还不存在，那么它将被加入.INI 文件中。MSVC 描述一门工具需要5个内容：出现在菜单上的名字(ToolMenuTextn)，.EXE 文件的路径名(ToolPathNamen)，可能会用到的运行时参数(ToolArgumentsn)，可能会用到的启动目录(ToolInitialDirn)。每次运行工具时是否要向用户征求参数(ToolAskArgumentsn)。在子过程中，这些内容的更新或设置是这样完成的：

```
Private Sub Install_MSVC150()
IniDatal. FileName=" msvc. ini. "
IniDatal. Section=" Microsoft Visual C ++"

Dim N As String
N=GetToolNumber_MSVC150("&"+App. Title)

IniDatal. KeyName=" ToolPathName" +N
IniDatal. Text=MakePath("",App. Path,App. EXEName,". exe")
IniDatal. KeyName=" ToolMenuText" +N
IniDatal. Text=" &" +App. Title
IniDatal. KeyName=" ToolArguments" +N
IniDatal. Text=" /IDE=MSVC150"
IniDatal. KeyName=" ToolInitialDir" +N
IniDatal. Text=" $Dir"
IniDatal. KeyName=" ToolAskArguments" +N
```

```
IniDatal.Text=" 0"

IniDatal.KeyName=" Includel"
If InStr(IniDatal.Text,VBAPI_Path)=0 Then
    IniDatal.Text=IniDatal.Text+";" +VBAPI_Path
End If
IniDatal.KeyName=" Lib1"
If InStr(IniDatal.Text,VBAPI_Path)=0 Then
    IniDatal.Text=IniDatal.Text+";" +VBAPI_Path
End If

MsgBox App.Title+" has been installed into Microsoft Visual C ++1.5x.Please restart
for it to become available as part of the Tools menu.", vbInformation
End Sub
```

ToolInitiaiDirn 的“$Dir”值是 MSVC 提供的宏，它将在工具运行的时候展开。MSVC 中有很多这样的宏。

我们最后与 MSVC 打交道是把 Visual Basic CDK 的目录加入 MSVC 的搜索路径中。MSVC 保存当前的搜索路径及此前用到的三条搜索路径。不必关心前面的路径，我们只需修改当前的搜索路径：Includel 及 Libl。

10.3.1 创建全新的 MSVC 工程

要让 IDE 创建一个新的工程，最简洁的方法是什么?

应该是 DDE(动态数据交换)。然而 Microsoft 或 Borland 都未公布他们各自的 C++IDE 的 DDE 接口。

那么，发送击键给 IDE，怎么样?

Borland 的 IDE 可以用这种方法。Microsoft 也理应这样。毕竟 Visual Basic、SendKeys 语句以及 Windows，都是他们自家的发明创造。然而，这样做困难重重；使用“%O”击键并不能如愿地关闭 Edit Project 对话框。解决这一问题可以用 tab 键跳到 Close 按钮，再发送回车键；然而还有一个更为严重的问题。

在 MSVC 中，当创建新的工程时，复选框“Use Microsoft Foundation Classes”将继承前一工程的设置。通过发送击键，可翻转其值。然而，要得到它的当前值的确不容易。另外还有几个 Project 选项也是这样。

如此看来，至少对 MSVC 来说，创建工程最容易的办法就是老老实实地生成一个 MAK 文件。

幸而，这对我们来说是很容易的。我们已经有了生成代码的整个机制；只需加入一些替换体就万事大吉。这件工作如此简单，究其原因，是因为最困难的部分，即创建.MAK 文件的框架、MSVC 可以帮我们完成。

由 MSVC 生成的.MAK 文件指定了工程名、目标类型、内存模式和构件。我仿造了一个这样的文件，并把其中的硬编码元素替换成 VBXGEN 风格的替换体。经过改造后，我得到了下面的文件 MAIN.MAK，并把它与其他框架代码文件一起保存在资源目录当中：

```
# Microsoft Visual C ++generated build script-Do not modify

PROJ= $PROJECT
DEBUG-1
PROGTYPE=2
CALLER=
ARGS=
DLLS=
D_RCDEFINES=-d_DEBUG
R_RCDEFINES=-dNDEBUG
ORIGIN=MSVC
ORIGIN_VER=1.00
PROJPATH= $DIRECTORY\
USEMFC=0
CC=cl
CPP=cl
CXX=cl
CCREATEPCHFLAG=
CPPCREATEPCHFLAG=
CUSEPCHFLAG=
CPPUSEPCHFLAG=
FIRSTC=
FIRSTCPP=
RC=rc
CFLAGS_D_VEXE= /nologo /W3 /FR /G2 /Zi /D_DEBUG /Zp /Od /GD /ALw /Gc /
Fd" TEST16.PDB"
CFLAGS_R_VEXE= /nologo/W3/FR/G2/DNDEBUG/Gs/Zp/Ox/GD/ALw/Gc
LFLAGS_D_VEXE =/NOLOGO/ONERROR: NOEXE/NOD/PACKC: 61440/CO/
ALIGN: 16
LFLAGS_R_VEXE=/NOLOGO/ONERROR: NOEXE/NOD/PACKC: 61440/ALIGN:
16
LIBS_D_VEXE=oldnames libw commdlg shell olecli olesvr ldllcew
LIBS_R_VEXE=oldnames hbw commdlg shell olech olesvr ldllcew
RCFLAGS= /nologo
RESFLAGS= /nologo
RUNFLAGS=
DEFFILE=MAIN.DEF
OBJS_EXT=
LIBS_EXT=vbapi.lib
!if" $(DEBUG)" ==" 1"
```

```
CFLAGS= $ (CFLAGS_D_VEXE)
LFLAGS= $ (LFLAGS_D_VEXE)
LIBS= $ (LIBS_D_VEXE)
MAPFILE=nul
RCDEFINES= $ (D_RCDEFINES)
!else
CFLAGS= $ (CFLAGS_R_VEXE)
LFLAGS= $ (LFLAGS_R_VEXE)
LIBS= $ (LIBS_R_VEXE)
MAPFILE=nul
RCDEFINES= $ (R_RCDEFINES)
!endif
!if[if exist MSVC. BND del MSVC. BND]
!endif
$ IFDEF $ LANGUAGE_C
SBRS=HELP. SBR\
        MAIN. SBR\
        VISUAL. SBR
#ENDIF
$ IFDEF $ LANGUAGE_CPP
SBRS=VBXXX. SBR\
        MAIN. SBR\
        PORTAGDI. SBR
#ENDIF
$ IFDEF $ LANGUAGE_C
HELP_DEP=internal. h
MAIN_DEP=internal. h
VISUAL_DEP=internai. h
$ ENDIF
$ IFDEF $ LANGUAGE_CPP
VBXXX_DEP=vbxxx. h porlagdi. h
PORTAGDI_DEP=portagdi. h
MAIN_DEP=vbxxx. h
$ ENDIF

MAIN_RCDEP=main. ico

all: $ (PROJ). VBX $ (PROJ). BSC

$ IFDEF $ LANGUAGE_C
```

```
HELP.OBJ: HELP.C $(HELP_DEP)
    $(CC) $(CFLAGS) $(CUSEPCIIFLAG)/c HELP.C

MAIN.OBJ: MAIN.C $(MAIN_DEP)
    $(CC) $(CFLAGS) $(CUSEPCHFLAG)/cMAIN.C

VISUAL.OBJ: VISUAL.C $(VISUAL_DEP)
    $(CC) $(CFLAGS) $(CUSEPCHFLAG)/cVISUAL.C

MAIN.RES: MAIN.RC $(MAIN_RCDEP)
    $(RC) $(RCFLAGS) $(RCDEFINES)-r MAIN.RC
$ENDIF

$IFDEF $LANGUAGE_CPP
VBXXX.OBJ:    VBXXX.CPP $(VBXXX_DEP)
    $(CC) $(CFLAGS) $(CUSEPCHFLAG)/cVBXXX.CPP

MAIN.OBJ: MAIN.CPP $(MAIN_DEP)
    $(CC) $(CFLAGS) $(CUSEPCHFLAG)/cMAIN.CPP

PORTAGDI.OBJ: PORTAGDI.CPP $(PORTAGDI_DEP)
    $(CC) $(CFLAGS) $(CUSEPCHFLAG)/cPORTAGDI.CPP

MAIN.RES: MAIN.RC $(MAIN_RCDEP)
    $(RC) $(RCFLAGS) $(RCDEFINES)-rMAIN.RC
$ENDIF

$(PROJ).VBX:: MAIN.RES

$IFDEF $LANGUAGE_C
$(PROJ).VBX:: HELP.OBJMAIN.OBJ VISUAL.OBJ $(OBJS_EXT) $(DEFFILE)
    echo> NUL@ < < $(PROJ).CRF
HELP.OBJ+
MAIN.OBJ+
VISUAL.OBJ+
$(OBJS_EXT)
$(PROJ).VBX
$(MAPFILE)
d:\msvc\lib\+
d:\msvc\mfc\lib\+
```

```
$(LIBS)
$(DEFFILE);
〈〈
    link $(LFLAGS)@$(PROJ).CRF
     $(RC)$(RESFLAGS)MAIN.RES$@
    @copy $(PROJ).CRF MSVC.BND
$ENDIF
$IFDEF $LANGUAGE_CPP
$(PROJ).VBX:: VBXXX.OBJ MAIN.OBJ PORTAGDI.OBJ $(OBJS_EXT) $(DEF-
    FILE)
    echo〉NUL@〈〈$(PROJ).CRF
VBXXX.OBJ+
MAIN.OBJ+
PORTAGDI.OBJ+
$(OBJS_EXT)
$(PROJ).VBX
$(MAPFILE)
d:\msvc\lib\+
d:\msvc\mfc\lib\+
$(LIBS)
$(DEFFILE);
〈〈
    link $(LFLAGS)@$(PROJ).CRF
     $(RC)$(RESFLAGS)MAIN.RES$@
    @copy $(PROJ).CRF MSVC.BND
$ENDIF
$(PROJ).VBX:: MAIN.RES
    if not exist MSVC.BND $(RC)$(RESFLAGS)MAIN.RES $@

run: $(PROJ).VBX
     $(PROJ)$(RUNFLAGS)


$(PROJ).BSC: $(SBRS)
    bscmake@〈〈
/o$@$(SBRS)
〈〈
```

对那些新增和改换过的地方我都作了明显的标示。

总共有4个替换体：$MSVC150、$LANGUAGE_C、$LANGUAGE_CPP 和

$ DIRECTORY。当装入 Prime 并调用 SetTargetIDE 时，第一个替换体的值被设定，这个过程是在 CodeGenerator 表中定义的：

```
Public Sub SetTargetIDE()
Dim Buffer As String, i As Integer
i=InStr(UCase $ (Command $ )," /IDE=")
Ifi> 0 Then
        Buffer=Mid $ (Command $ ,i+5)
        i=InStr(Buffer,"")
        Ifi> 0 Then Buffer-Left $ (Buffer,i-1)
        TargetIDE=UCase $ (Buffer)
End If
End Sub
```

TargetIDE 是 CodeGenerator 的静态变量，在后面决定生成哪个 IDE 工程的时候要用到这个变量。余下的替换体是在Prime 的OK 按钮的Click 事件处理程序中，与其他变量值一道设定的。

有了这些信息，生成 MSVC 工程就变得相当简单了。用户选中 Tools...VB Genie 运行 VBX Genie，然后填写几页信息，再单击 OK 按钮。工程文件.MAK 就同其他文件一道被创建出来。如果有一个打开的工程，VBX Genie 最后还应该关闭它，然后打开新工程。这一函数在 CodeGenerator 模块中：

```
Private Function AddToProject_MSVC150() As Boolean
Dim Window As Integer
Window=FindApplication("Microsoft Visual C++")
If Window <> 0 Then
        SetActiveWindow Window
        'File..Save All
        SendKeys"%FV", True
        'Window..Close All
        SendKeys"%WA", True
        'Project..Close
        SendKeys"%PC", True
        'Project..Open <project name> <Enter>
        SendKeys "%PO" +_
                Brace (MakePath ("", Prime.txt_Directory, Prime.txt_Name,""))
                +_
                " {Enter}", True
End If
AddToProject_MSVC150=(Window<>0)
End Function
```

函数 Brace 用来应付长文件名当中用到的新字符。这些字符对 SendKeys 语句有着特殊含

义，所以它们必须以括号隔开：

```
Public Function Brace(ByVal Text As String)
Dim Braced As String, i As Integer, c As String
For i = 1 To Len(Text)
    Select Case Mid $ (Text,i,1)
    Case " i"," ^ ","%"," ~"," (",")"," [","]"
        Braced=Braced+" {" +Mid $ (Text,i,1)+"}"
    Case Else
        Braced=Braced+Mid $ (Text,i,1)
    End Select
Next i
Brace=Braced
End Function
```

现在如果用户要创建工程，应该不会有什么问题了。

10.4 把 VBX Genie 安装到 Borland C++

要是安装 Borland IDE 工具有安装 Visual C++工具那么简单就好了！然而 Borland 公司，由于成功地推出系列化的对象产品，于是就把它的环境参数一股脑地保存在一个满是 C++对象的、其他任何人既读不了又写不了的文件中。(既然这样，何必要有一个.INI文件呢?)

好在 Borland 公司用于描述新工具的对话框可以正确对 Visual Basic 的 SendKeys 作出反应。条件是找到 Borland C++ IDE，启动它，并把它锁定在主窗口。

利用公用对话框控件中的 Open File 对话框，InstallTool 表可以让用户确定 BCW.EXE 所在的位置：

```
Private Function StartBCW() As Integer
StartBCW=0

CmDialogl.CancelError=True
CmDialogl.DefaultExt=" exe"
CmDialogl.DialogTitle=" Locate Borland C++ IDE"
CmDialogl.FileName=" bcw.exe"
CmDialogl.Filter=" Borland C++|bcw.exe|Programs|*.exe"
CmDialogl.FilterIndex=1
CmDialogl.Flags=cdlOFNHideReadOnly Or cdlOFNFileMustExist

On Error Resume Next
CmDialogl.ShowOpen
If Err=0 Then
```

```
        Dim x As Integer, Window As Integer
        x=Shell(CmDialog1.FileName,vbMaximizedFocus)
        For x=1 To 100
                Window=FindApplication("Borland C++")
                If Window<>0 Then Exit For
        Next x
        StartBCW = Window
    End If
    End Function
```

如果调用失败，函数StartBCW返回0，否则返回Borland C++的主窗口句柄。定位BC++主窗口的任务由函数FindApplication完成，它所封装的Windows API调用能够根据给定的窗口标题部分把对应的窗口找出来。WinAPI.BAS模块处理本地Windows API调用，函数StartBCW就是这个模块的一部分：

```
Public Function FindApplication(ByVal Text As String)As Integer
Dim Window As Integer, Buffer As String
Window=GetWindow(Prime.hWnd,GW_HWNDFIRST)
While Window <> 0
    Buffer=Space$(GetWindowTextLength(Window)+1)
    GetWindowText Window, Buffer, Len(Buffer)
    IfLeft$(Buffer,Len(Text)) =Text Then
        FindApplication=Window
        Exit Function
    End If
    Window=GetWindow(Window,GW HWNDNEXT)
Wend
FindApplication=0
End Function
```

GetWindows是一个Windows API函数。给定某个窗口的句柄(比如说我们自己的主窗口Prime)，就可利用这个函数逐个遍历同一层次全部窗口所组成的列表。GetWindowTextLength和GetWindowText两个函数从我们检索的每个窗口中取出文本。

这些API(以及我们快要用到的其他API)的Declare语句已被复制到WINAPI.BAS的说明部分：

```
Option Explicit

Declare Function GetWindow Lib" User"
    (ByVal hWnd As Integer,ByVal wCmd As Integer)As Integer
Global Const GW_HWNDFIRST=0
Global Const GW_HWNDNEXT=2
```

```
Declare Function GetWindowTextLength Lib" User" (ByVal hWnd As Integer)As Integer
Declare Sub GetWindowText Lib" User" (ByVal hWnd As Integer,ByVal lpString As
String,ByVal aint As Integer)

Declare Sub SetActiveWindow Lib" User" (ByVal hWnd As Integer)
```

现在我们已经有了函数 StartBCW，它找到 Borland C++并装入它，再返回其主窗口句柄。在这里，已经完成了相当一部分工作！

下面的任务是向 IDE 发送适当的击键以加入这个新工具。击键序列"<alt>OT"将激活 Options. Tools 命令并打开 Tools 对话框。对话框罗列出所有已安装的工具。其外观如图10.3。

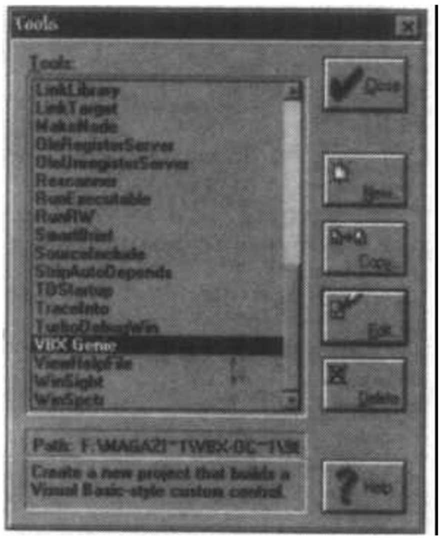

图10.3 Tools 对话框

好消息说完了。还有一条坏消息等着我们领教。那就是，要用编程的方法判断 VBX Genie 是否在对话框的工具列表中并非易事。

然而办法还是有的。我们得弄清楚是要加入一个新的工具呢?还是编辑一个已经有了的工具?这又得求助于 Windows API。

好在 Borland C++自带了一个相当不错的工具 WinSight。Winsight 在屏幕上列出所有的窗口(包括隐藏的窗口)，窗口以提纲的形式组织起来，还显示了一些有用的信息，并且允许您深入地查询它们。

在显示上面的 Tools 对话框时，Winsight 给出了图10.4的信息。

真是绝境逢生!我们可以发现，Tools 对话框是一个弹出式窗口(这没什么奇怪)，列表框其实是一个普通的列表框。而且，当高亮度显示它(如图10.4所示)并按 ENTER 键，我们还可以知道更多，如图10.5。

任凭窗口句柄变幻不定，窗口 ID 却始终如一。所以我们要做的只是：

1. 凭 Tools 对话框的标题取得 Tools 对话框的句柄，其方法同我们定位 Borland C++主窗口。

2. 凭窗口 ID，调用 API 函数 GetDlgItem 取得列表框的句柄。

3. 标准列表框消息结合 SendMessage 函数，查询列表框项，以期找到所要的项。如果找到，

图10.4　用 WinSight 显示窗口信息

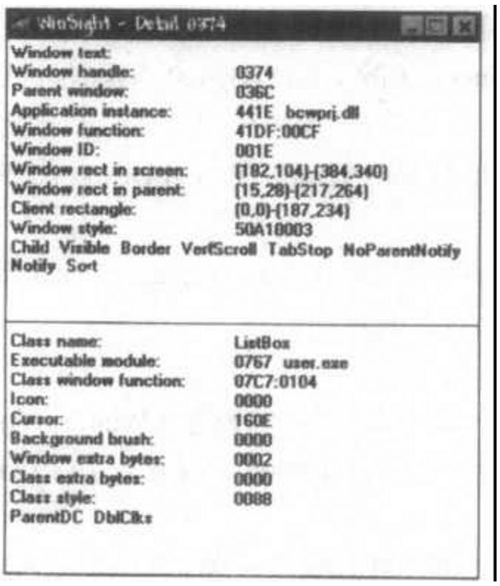

图10.5　用 WinSight 显示更多的细节

用程序实现按动 Edit 按钮的动作；如果没有找到，则按动 New 按钮。

下面是代码：

```
Private Sub OpenBCWToolEdit(ByVal Window As Integer)
SetActive Window Window
DoEvents
'Options.. Tools
SendKeys"%OT", True
DoEvents

Window=FindWindow("BorDlg_Gray","Tools")
If Window=0 Then Error 31024
Window=GetDlgItem(Window,&HIE)
If Window=0 Then Error 31024

Dim i As Long
i=SendMessage(Window,LB_FINDSTRINGEXACT,-1,(App,Title))
```

```
Ifi>-1 Then
        i=SendMessage(Window,LB_SETCURSEL,i,"")
        SendKeys"%E", True
Else
        SendKeys"%N", True
End if
DoEvents
End Sub
```

在这个子过程中，有三处 Windows API 调用是我们以前没见过的：FindWindow，GetDlgItem 和 SendMessage。我从 Visual Basic 的 API Declares 帮助文件中得到这些说明，但在把它们加入 WINAPI.BAS 的说明部分之前作了些小的调整：

```
Declare Function FindWindow Lib"User" (ByVal ClassName As String,ByVal WindowCaption As String)As Integer
Declare Function GetDlgItem Lib" User" (ByVal Dialog As Integer,ByVal ItemID As Integer)As Integer
Declare Function IsChecked Lib"User"Alias"IsDlgButtonChecked"(ByVal hDlg As Integer,ByVal nIDButton As Integer)As Integer

Declare Function SendMessage Lib" User" (ByVal Window As Integer,ByVal MsgID As Integer,ByVal wParam As Integer,ByVal lParam As String)As Long
Const WM_USER=&H400
Global Const LB_FINDSTRINGEXACT=(WM_USER+35)
Global Const LB_SETCURSEL=(WM_USER+7)
```

VB4中预先提供的那些 Declare 语句比 VB3中的要好，然而也并非尽善尽美。FindWindow 寻找一个窗口类和窗口标题都匹配的窗口(而且标题必须从头至尾精确匹配)。在 API Text Viewer 中，这个 API 函数的两个参数列出的都是“Any”，这可以满足任何参数为 NULL(0)的情况。但是，“Any”并不适用于 Visual Basic 的 String 向 C 风格的字符串转换。

SendMessage 也面临类似问题；对不同的消息来说，第四个参数 lParam 所对应的数据类型可以大相径庭。好在我们要用的只有一种数据类型(还是 String)。

我们要发送的两个消息 LB_FINDEXACTSTRING 和 LB_SETCURSEL，都基于 WM_USER，所以这三个常量都必须包括进去。

窗口定位函数调用失败时返回的错误号 ERROR，是预定义错误表中的一个。错误号31024来源于 OLE，意思是“Unable to create link”(不能创建连接)。用户应能明白，这里的连接指的是与 Borland C++的连接。我不知道什么原因会导致这个问题，但不管怎样，都别指望能在程序中排除它！

现在出现了 Tool Options 对话框需要我们来填充。图10.6是对话框填入数据后的样子。

下面的代码是把相应的击键发送给对话框：

```
Private Sub FillInBCWToolOptions()
```

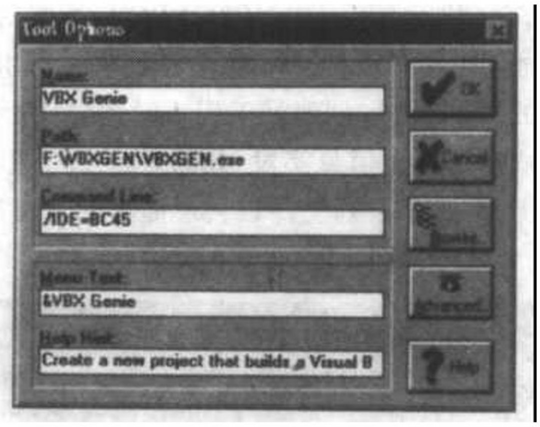

图10.6 Tool Options 对话框

```
SendKeys"%N" +Brace(App.Title), True
DoEvents
SendKeys"%P" +Brace(MakePath("",App.Path,App.EXEName,"exe")), True
DoEvents
Sendkeys"%C/IDE=BC45", True
DoEvents
SendKeys"%M&" +Brace(App.Title), True
DoEvents
SendKeys"%HCreate a new project that builds a Visual Basic-style custom control", True
DoEvents
'Close the Tool Options dialog
SendKeys" {ENTER}", True
DoEvents
'Close the Tools dialog
SendKeys"%C", True
DoEvents
End Sub
```

到这里，我们已经有了足够的资本来构建 Install_BC45：

```
Private Sub Install_BC45()
Dim Window As Integer
Window=StartBCW()
OpenBCWToolEdit Window
FillInBCWToolOptions
SendKeys"%FX", True'Close BCW
MsgBox" VBX Genie has been installed into Borland C++'Tools menu.", vbInformation
End Sub
```

10.4.1 创建全新的 Borland C++工程

跟安装 VBX Genie 一样，在 Borland C++中创建新工程要比在 Microsoft Visual C++中

复杂，原因是它必须完全由 SendKeys 来完成。还好，BCW 能够对击键作出正确响应！我只能说，绝大多数情况是这样。有一个不肯合作——还是碰到它时再讲吧。

与 Visual C++一样，在 VBXGenie 创建好工程目录和文件之后，接下来需要您给出 Project..New 命令，并填写对话框的各个信息字段。此时，这个标题为“New Target”的对话框所需要的信息不仅仅是构成工程的一系列文件；它还需要知道类库(如 OWL)的位置。图10.7是对话框的外观。

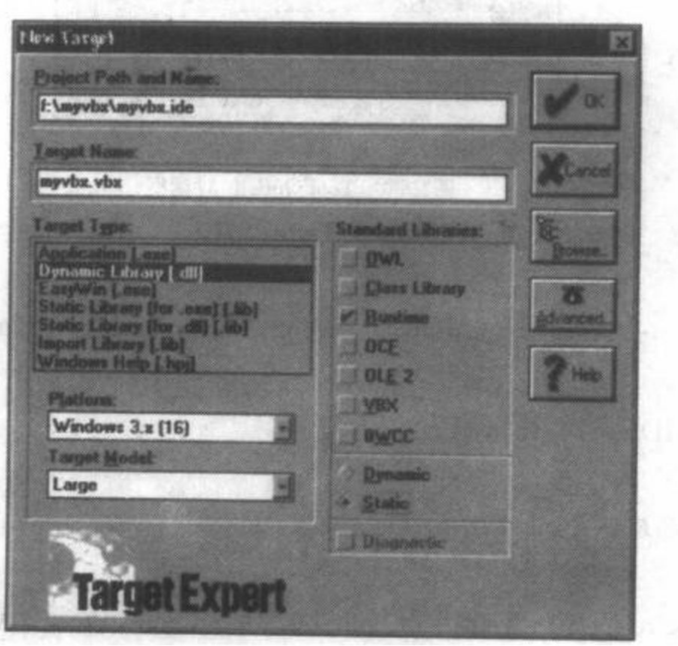

图10.7 New Target 对话框

问题马上就来了：我们必须取消两个复选框(OWL 和 Class Library)。不论复选框原有状态如何，我们希望取消复选框的复选状态，但是 Microsoft 公司从未提供用发送击键来实现这一点的方法。在程序中用上 CheckDlgButton 这个 Windows API 函数应该能行。只是，我的尝试没有成功！严格地说，这些并不是标准的复选框；它们是 BorCheck 控件，是 BWCC 定制控件集的一部分。它们理应与标准复选框一样地工作；因而当它们并不象“理应”那样时，就尤其令人沮丧了。

好在我的最后一搏是成功的：用 IsDlgButtonChecked 函数查看它是否是复选状态，若是，就发送击键取消复选。(即使这样也并不是一帆风顺的：起初它并不好好配合……直到我又把各个信息字段的次序又重排一遍！嘘！——)

关闭对话框之前，必须按动 Advanced 按钮再填写一个对话框。我不明白它为什么叫做“Advanced”(高级选项)，“Initial nodes”(初始节点)倒是更名副其实，见图10.8。

我们想要关闭源节点(选中“No Source Node”)，还要取消.rc 和.def 的复选状态。这是因为，VBX Genie 已经生成了那些文件，我们希望手工地把它们加到工程里。

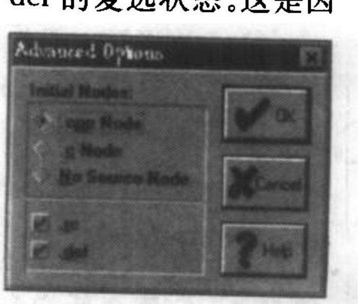

图10.8 在 Advanced Options 对话框中选定的选项

最后还要修改project(工程)选项。这个对话框是由 Options..Project 菜单命令激活的。我们得修改头文件和库的搜索路径；还要关闭两个 Case Sensitive(大小写敏感)连接选项，如图10.9所示。

看，又是一些复选框。

下面就是完成这些工作的代码。它疯狂地使用 SendKeys，以及 Find Window 和 IsChecked，后者是 IsDlgButtonChecked 的别名：

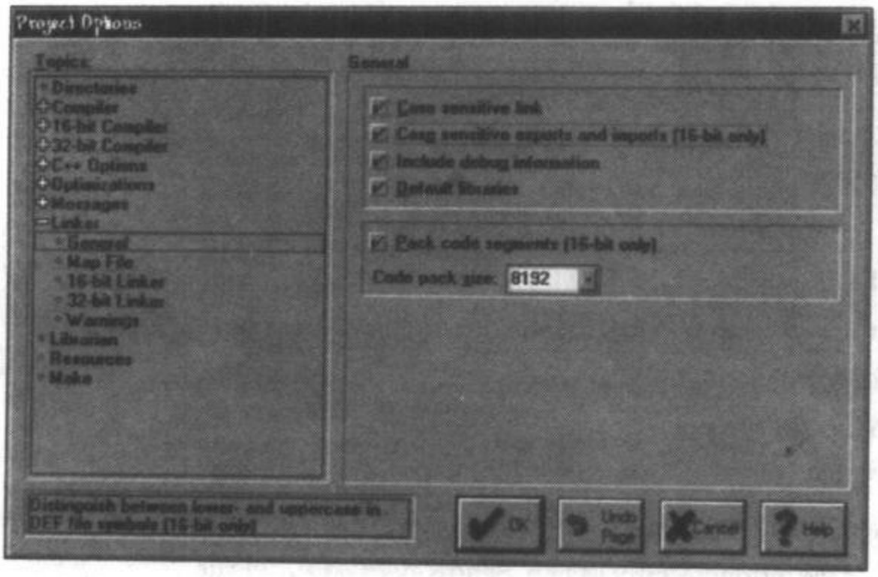

图10.9 Project Options 对话框

```
Function AddToProject_BC45()As Boolean
Dim Window As Integer
Window=FindApplication("Borland C++")
If Window <> 0 Then
   SetActive Window Window
   'Window..Close All..Windows
   SendKeys"%WAW", True
   'File..Save All
   SendKeys"%FL", True
   'Project..Close
   SendKeys"%PL", True
   'Project..New <project name> <Enter>
   SendKeys"%PN", True
   'Turn off OWL, Class Library, choose static library
   Window=FindWindow (" BorDlg_Gray", New Target")
   If IsChecked(Window,&H2775)Then SendKeys"%O", True
   If IsChecked(Window,&H2776)Then SendKeys"%C", True
   SendKeys"%S", True
   'Project path and name
   SendKeys"%P" +Brace(MakePath("",Prime.txt_Directory,_,
   Prime.txt_Name,"")+" .ide"), True
   'Target name
   SendKeys"%T" +Brace(Prime.txt_Name+".vbx"), True
   'Target type: DLL
   SendKeys"%YD", True
   'Platform: Win16
   SendKeys"%L{Home}", True
```

```
'Target model: Large
SendKeys" %ML", True
'Open Advanced dialog
SendKeys" %A", True
Do
        DoEvents
        Window=FindWindow("BorDlg_Gray","Advanced Options")
Loop Until Window <> 0
'Don't produce initial nodes
SendKeys" %N", True
If IsChecked(Window,&H68)Then SendKeys" %R", True
If IsChecked(Window,&H69)Then SendKeys" %D", True
SendKeys" {Enter}", True
'Close New Target dialog
SendKeys" {Enter}", True
'Add nodes to the project
SendKeys" {Insert}", True
SendKeys" vbapi.lib{Enter}", True
SendKeys" {Up}{Insert}", True
SendKeys" main.def{Enter}", True
SendKeys" {Up}{Insert}", True
SendKeys" main. rc {Enter}", True
Select Case Language
Case" C"
        SendKeys" {Up}{Insert}", True
        SendKeys" help.c{Enter}", True
        SendKeys" {Up}{Insert}", True
        SendKeys" visual.c{Enter}", True
        SendKeys" {Up}{Insert}", True
        SendKeys" main.c{Enter}", True
Case" C++"
        SendKeys" {Up}{Insert}", True
        SendKeys" portagdi.cpp{Enter}", True
        SendKeys" {Up}{Insert}", True
        SendKeys" vbxxx.cpp{Enter}", True
        SendKeys" {Up}{Insert}", True
        SendKeys" main.cpp{Enter}", True
End Select
'Display Project Options Dialog
SendKeys" %OP", True
```

```
        DoEvents
        'goto top of list
        SendKeys"%T^ {Home}", True
        'copy the include path
        SendKeys"%I^ {Insert}", True
        Dim Buffer As String
        Buffer=Clipboard.GetText()
        If InStr(Buffer,VBAPI_Path)=0 Then
                Buffer=Brace(Buffer+";"+VBAPI_Path)
                SendKeys Buffer, True
        End if
        SendKeys"%B^ {Insert}", True
        Buffer=Clipboard.GetText()
        If InStr(Buffer,VBAPI_Path)=0 Then
                Buffer=Brace(Buffer+";"+VBAPI_Path)
                SendKeys Buffer, True
        End If
        Clipboard.Clear
        'Display Linker general options page
        SendKeys"%T^ {Home}{Down 7}{+}{Down}"
        'Get dialog's window handle,
        'General Linker Options page
        Window=FindWindow("BorDlg_Gray","Project Options")
        Do
                DoEvents
                Window-GetDlgItem(Window,&H7D3)
        Loop Until Window <> 0
        'Set" Case Sensitive" options to unchecked
        If IsChecked(Window,&H20)Then SendKeys"%C", True
        If IsChecked(Window,&H21)Then SendKeys"%E", True
        'Close the Project Options dialog
        SendKeys" {Enter}", True
    End If
    AddToProject_BC45=(Window <> 0)
    End Function
```

真是一项艰巨的任务，对吧?但是有了这代码，VBX Genie 作为一种工具，在 Borland C++下就和在 Microsoft Visual C++下工作得一样好了。一个巨大的难题!但同时又是了不起的胜利。

至此，VBX Genie 大功告成!

第十一章　为 VBX Genie 创建安装程序

11.1　VBX Genie 安装过程记事

当我试图运行 Visual Basic 4的16位 Setup Wizard 以便生成 VBX Genie 的分布盘的时候，怪事接二连三地发生了。我试着把 VBX Genie 安装到各种机器上，结果太令人沮丧了！我发现安装工作有的正常，有的不行。更糟糕的是，当安装失败时，它们常常置计算机于死机的境地。

多亏有 CompuServe 上的 MSBASIC 论坛，使我能够得知其他程序员也遇到过同样的问题，并从他们那里得到一些线索。Microsoft 公司常驻此论坛的技术支持人员帮不了什么忙，对他们来说，Setup Wizard 永远都能完美地工作！但这确实是一个路子。我开始留意其他程序员遇到的类似问题的细节，并逐渐形成了一个解决办法，我将与您一同分享这个方法。

11.2　是平台，还是沙洲？

在我写这本书时，有四种 Windows 平台正被使用，然而只有两种仍在发行。它们是 Windows 95和 Windows NT，均为 Microsoft 竭力提倡；而 Windows3.1和 Windows for Workgroups 3.11最近已经失宠。OLE（对象链接与嵌入）是在 Windows3.1和 Windows for Workgroups 3.11发售时介绍给用户的；在其最初版本和补丁版本中，还存在着一些错误，而用以实现 OLE 的 DLL（动态连接库）却如雨后春笋般大量出现。Window 3.1/3.11其实并不带有 OLE，而 Microsoft 近期的所有应用程序（如 Microsoft Office）则都带有 OLE；这样，大多数——但不是全部！——运行 Win31的计算机上都有 OLE 的 DLL。

事实上，Windows 95带有大部分的 OLE DLL，但也不是全部。（其余的随 Visual Basic 4.0一起安装）。现在问题就来了：Windows 95的 OLE DLL 与 Windows 3.1/3.11不能兼容！

如果您在 Win31平台上安装了 Visual Basic4，Setup Wizard 将利用 Win31的 DLL 创建应用程序分布盘。这样，应用程序可以安全地安装到另一台 Win31机上。如果试图把应用程序安装到一般的装有 Microsoft Office 的 Win95机器上，也可以成功安装；因为分布盘上的旧版本 DLL 不会被拷贝到目标机上。然而，如果这台 Win95计算机还没有安装新的 OLE 应用程序（比如 Visual Basic 4.0或 Microsoft Office 95），它就会缺少某个 OLE DLL——缺少的这个将由分布盘提供。但是，分布盘提供的 DLL 会与其他更新版本的 OLE DLL 们发生冲突，原因是所有这些 DLL 必须配套使用！

如果您是在 Win95机上安装了 Visual Basic 4.0，那么您用 Setup Wizard 创建了应用程序分布盘后，如果把应用程序安装到 Win31机器时往往会令目标机死机，究其原因，是 Wine95的 OLE DLL 比目标机上的版本更高，因此会改写它们。

我决心要找到一个解决办法。我的机器上安装了 Win95和 VB4，于是我检查了\VB\SETUPKIT\KITFILES\SYS16中的所有文件（它们是在 VB 安装时从\WINDOWS\SYSTEM 中拷贝得到的。这可真算得上是一个有趣的方法，如果您随后对 DLL 作了升级，那么在 SYS16中的这些文件不就过期失效了吗）。在这些文件里，我筛选出不是来自于 Windows 95分布盘的文

件。(不错，我有软盘版的 Win95)除了四个文件，所有其他文件都是随 Windows 95安装的，因此把\VB\SETUPKIT\KITFILES\SYS16中的所有 Windows 95的文件替换成 Windows3.1的文件，整个问题几乎就解决了……除了那另外的四个文件。对于四个文件之外的其他文件，在安装到 Windows 3.1机器时，如果机器里没有这些文件，或者已经过期，安装程序就安装它们；在安装到 Windows 95机器时，安装程序会发现机器里已有这些文件的更高版本，从而不会改写它们。

剩下的四个文件是：

- OLE2.REG
- OLE2PROX.DLL
- SCP.DLL
- STDOLE.TLB

这才是问题的症结之所在。这些名字的文件，它们并非来自 Windows 95，当然也非来自 Win31。

解决办法是在分布盘上保留这四个文件的两套版本(用不同的文件名)；制作两个版本的 SETUP.LST(Setup 将读取这个文件以决定要拷贝些什么到目标机上去)；以及在 SETUP.EXE 中加入一个预处理步骤，先检测是安装到 Win31还是 Win95上，然后相应地改换.LST 的文件名，以使标准的安装程序能够正常运行。

这看起来似乎很混乱?肯定是!任何混乱的事情注定是错多对少。所以，我们最好把这个过程自动化。

11.3 Setup Wizard 都做些什么?

在编写我们的 Setup Wizard Workaround(Setup Wizard 设置程序)和 Workaround Installer(设置程序的安装程序)之前，我们应该先看看在没有 Setup Wizard 设置程序时，Setup Wizard 都做了些什么。

Setup Wizard 所做的工作可分为七步。第一步，指定想要发行工程的 VB 工程文件名。第二步，指定发行文件应存放的位置，您可以选择软盘驱动器(包括磁盘密度规格)或者硬盘上的目录。然而，您不能把目录分割为“软盘”大小的逻辑块；如果最终的发行媒介是软盘的话，您就得现在声明。第四步，列出您的应用程序可能引用到的所有 OLE 服务程序，比如 Microsoft Word 或 Excel。在这一步的列表中不包括 OLE 控件，而且列表常常是空的。第五步，列出 OLE 控件。如果您的应用程序是一个 OLE 服务器程序，那么第六步的工作很重要。如果是 OLE 服务器程序，您也许希望把它和其他此类服务器程序放到同一个目录当中。否则，“Application Directioy(应用目录)”的默认值一般都是可行的。

第七步，指定您的应用程序要用到的所有文件部件。即使应用程序并不是一个 OLE 服务器程序或客户程序，您也会在这里发现一些 OLE DLL，这并不奇怪。Visual Basic 自身就是一个 OLE 服务器程序，甚至是运行态的 DLL，因此 OLE DLL 是必需的。这意味着，对于一个16位应用程序来说，如果在3.0版本下可以做到单张软盘上的话，现在则至少需要两张。(同样的应用程序，如果编译成32位的，那么也只需要一张分布盘，因为所有32位 Windows 平台都假定已经安装了所需的32位 OLE DLL。)这一步所引起的安装问题最大，让我们更深入地探讨一下。

11.3.1 更深入地探讨安装问题

在安装 Visual Basic 时，一个名为 SWDEPEND.INI 的文件被创建到 Windows 目录。(VB 的文档错误地将它引用为“SWDEPEND.TXT”，根本没有这个文件。)SWDEPEND.INI 包含与各个常用 VB 部件相关联的文件列表。例如，Crystal Reprots OCX 需要一大堆 DLL；下面指明了它的关联：

```
[CRYSTL16.OCX]
Dest=$(WinSysPath)
Register=$(DLLSelfRegister)
Uses1=CRPE.DLL
Uses2=CRXLATE.DLL
Uses3=uxddisk.dll
Uses4=uxfdif.dll
Uses5=uxfrec.dll
Uses6=uxfsepv.dll
Uses7=uxftext.dll
Uses8=pdbdao.dll
Uses9=pdctdao.dll
Uses10-pdirdao.dll
Uses11=pdsodbc.dll
Uses12=PDBBND.dll
```

顺便指出，上面的列表不正确(尽管它是由 VB 安装程序创建的)。我们的 Workaround Installer 也能改正这里的错误。这样做的好处是，您可以很容易地往.INI 文件里添加其他项。在安装一个新的 VBX 或 OCX 时，这可能是您安装工作的一部分。这样做使 Setup Wizard 自动确保对于使用了新近安装控件的任何应用程序，其所需的部件都与应用程序一起发行。

Setup Wizard 还用到文件 SETUPWIZ.INI。如果是在安装了 VB3.0后安装 VB4.0，您会发现 SETUPWIZ.INI 中有的项好象是复制了 SWDEPEND.INI 中的 DLL 关联，但是仔细看则会发现这些项只是针对 VBX。SETUPWIZ.INI 这方面的功能被 SWDEPEND.INI 取代了，但是 SETUPWIZ.INI 中的其他节(section)仍然发挥着作用。

还是回到安装的第七步；手工加入任何想要发行的文件：一般是一个帮助文件：经常加入的还有版权许可文件或者 README. TXT 文件。

您也许已经注意到，在上面的 SWDEPEND.INI 片断中，指定了默认的安装路径。而单击 File Details 按钮，可以按您的意图修改它。

在击中 Finish 按钮后，就开始了迄今为止我们遇到的时间最长的操作。这时，Setup Wizard 着手压缩所需的文件。为什么 Setup Wizard 不一次性地压缩 OLE DLL，并在 KITFILES\SYS16中保存压缩后的文件，而保存容易过期失效的原始文件呢?我不知道原因——他们这样设计的时候并没有征求我的意见。而结果是每次使用 Setup Wizard 都会轻易地在压缩工作上花费五到十分钟的时间。

这些事情完成之后，您会发现，在分布盘或目录中除了压缩了的文件，还有几个未压缩

的文件。一个是 SETUP.EXE，这个小型程序读取另一个文件 SETUP.LST，以判定需要拷贝哪些文件才能执行 SETUP1.EXE。SETUP1.EXE 是一个 Visual Basic 应用程序，它是几个未压缩文件中的主角!SETUP.EXE 的任务是显示一个小对话框。当 SETUP1.EXE 等文件被拷贝到目标硬盘之后，SETUP1.EXE 就取代 SETUP.EXE 继续工作。此时在屏幕上显示漂亮的阴影背景。这个程序还读取 SETUP.LST 以决定剩下的还有哪些文件需要拷贝、拷贝到哪里。然后展开剩余的这些压缩文件，添加程序组图标，再退出。

有必要对文件 SETUP.LST 作一番研究，因为在实现设置程序的过程中我们会大量地用到它。首先注意，尽管它的后缀是 LST，它却是个.INI 文件。这样，一般的 Windows“profile（文件初始化)”函数(或者我的 IniD ata 控件)都可以读取它或修改它。SETUP.LST 分为三节。Bootstrap(自举)节是第一节，它由 SETUP.EXE 读取：

```
[BootStrap]
File1=1,SETUP1.EX_, SETUP1.EXE,$(WinPath),$(EXESelfRegister),, 8/15/_
1995, 138144, 4.0.0.2422
File2=1,, VSHARE.38_, VSHARE.386, $(WinSysPath),,, 11/8/1994, 14933,
3.11.0.401
File3=1,, STKIT416.DL_, STKIT416.DLL, $(WinSysPath),,, 8/15/1995, 5120,
4.0.2422.0
```

描述一个文件要用到多达十个参数，以指明文件大小、日期、版本、以及文件名、是否跨越多张磁盘。这些信息有助于防止分布盘被弄乱。

Files 节的格式与 BootStrap 节相同：它包括了将由 SETUP1.EXE 展开的文件列表。剩下的 Setup 节的内容是 SETUP1.EXE 为实现安装用户化要用到的各种信息：

```
[Setup]
Title=Setup Wizard Workaround
DefaultDir=$(ProgramFiles)\SWWRKRND
Setup=SETUP1.EXE
AppExe=SWWRKRND.EXE
Btrieve=0
```

在 SETUP.LST 中，还有一个小项很有意思。它实在可算是一项庞大的内容。它是这样开始的：

```
; The following lines may be deleted in order to obtain extra
; space for customizing this file on a full installation diskette.
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

后面还是这些奇怪的东西。实际上它占用的空间是如此之大，以致 SETUP.LST 文件就占了至少 6K 的磁盘空间。请别忘了，Setup Wizard 是利用文件大小来决定磁盘布局的。Microsoft 公司此举是基于如下的考虑，即如果您想要往 SETUP.LST 文件中添加一点什么，您

可以删除这些填充行以获得空间。这对我们的工程是有益的，因为我们的工程中就要删除 SETUP. LST 文件并以两个文件取而代之——去掉了填充行，这两个文件就放得下了。

11.4 制订我们的“进攻”计划

当我遇到编程问题时，我从不抱怨。或使只抱怨一点点。但随即我深吸一口气，把这个问题看成是一个新工程，决心要象解决其他工程一样解决它。诚然，这它短视的修补更花时间，但确实是一种防微杜渐的做法。把问题当成工程来对待，由此得到的好处与经由正式设计和实现的工程是一样的；许多可重用的软件部件可让我在以后的工程里节约时间。

创建 Setup Wizard Workaround 需要两个主要部件：

●其中一个“genie”用来拷贝文件和改正其他的 Setup Wizard 失当之处。它在一台机器上只需运行一次。

●另一个“genie”对改正后的 Setup Wizard 输出文件作修改，把新的“预安装”可执行文件添加进去。每次使用 Setup Wizard 时都要运行它一次。

如果再作深入讨论，那么还有几个底层次的通用性部件。例如，同 Microsoft 的许多 Wizard 程序(包括 Setup Wizard)类似，这两个“genie”也可开发成多页风格的应用程序。所以为什么不首先创建一个普适的“Genie”模子，然后以此为基础创建两个 genie 呢?还有，第一个 genie 必须展开一些压缩文件。为此可创建一个可重用的“展开文件”的模块，以及“进度条”模块。好吧，倒满一杯咖啡：我们要忙起来了。

11.5 创建预安装程序

尽管在别的工程中可能用不到，但是我们仍然得创建“预安装”程序。这是个小型的 Windows 可执行程序，它的任务只是简单区分运行环境是 Win31还是 Win95，据此把 SETUP. 31 或 SETUP. 95改名为 SETUP. LST，然后启动原有的 SETUP. EXE(它得另改一个名字)。

我们把这个工程叫做 PRESETUP，而实际的可执行文件必须命名为 SETUP. EXE，因为这才是人们认同的名字；这个文件必须在原有的安装程序之前运行，原有的安装程序我们将改名为 SETUPA. EXE。

请不要把文件名混同于模块名。文件名是指磁盘上的可执行文件名；模块名是应用程序加载到内存后，Windows 用来引用这个应用程序的名字。您可以有许多同名然而不同内容的文件(当然，是在不同的文件夹里)；但是在同一时刻，Windows 只允许内存里有一个给定模块名的代码段。(您可以运行应用程序的几个副本，但是代码只加载一次。至少在16位 Windows 中是这样规定的。)

PRESETUP 必须尽量紧凑而且简单。为此它必须以 C 语言来写。由于程序是从软盘上运行，故各段必须预先装入。下面是程序的模块定义文件：

```
NAME        PRESETUP
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    1024
```

工程仅包括一个代码模块，它没有自己的头文件。代码模块命名为 MAIN. C，为简单起

见，我把它分为几个函数。先看看 WinMain()：

```
int PASCAL WinMain(HINSTANCE Instance,HINSTANCE PrevInstance,LPSTR CmdLine,int CmdShow)
    {
    ResetDiskFromPreviousUse();
    if(IsWindows95())
            Rename("SETUP.95","SETUP.LST");
    else
            Rename("SETUP.31","SETUP.LST");
    WinExec("SETUPA.EXE",SW_SHOWNORMAL),
    return 0;
    }
```

在阐明有什么需要重置后，我们将考虑一下函数 ResetDiskFromPreviousUse()。isWindows95()函数的功能是显而易见的；它决定对两个 SETUP.*中的哪个改名。WinExec()函数启动原有安装程序(它现在的名字是 SETUPA.EXE)。

ResetDiskFromPreviousUse()根据文件的缺失情况，把 SETUP.LST 改名为 SETUP.31 或 SETUP.95。这样，在终端用户对旧平台升级后，分布盘还可以继续使用：

```
static void near pascal ResetDiskFromPreviousUse(void)
    {
    OFSTRUCT of,
    if(OPenFile("SETUP.31,"&of,OF EXIST)==HFILE_ERROR)
        Rename("SETUP.LST","SETUP.31");
    if(OpenFile("SETUP.95",&of,OF_EXIST)==HFILE_ERROR)
        Rename("SETUP.LST","SETUP.95");
    }
```

OpenFile()是一个 Windows 函数，用它可以方便地检查某个文件是否存在。在 WinMain()中调用的 Rename()函数则是我自己写的：

```
static void near pascal Rename(char * OldName,char * NewName)
    {
    if(rename(OldName,NewName)!=0)
        {
        int Result=MessageBox(NULL,"Unable to write to setup disk. Please release
        the write-protect tab and retry.",
            " setup",
            MB_RETRYCANCEL|MB_ICONEXCLAMATION);
if(Result==IDRETRY)
            Rename(OldName,NewName);
        }
```

```
    }
```

函数 rename()请注意它与 Rename()在名称上的细微差别——来自于标准 C 库。因为调用 Rename()时，OldName 文件是实际存在的，所以 rename()调用不成功的原因是要么 OldName 文件被标记为只读，要么磁盘被写保护。Setup Wizard Workround 能把 SETUP. 31和 SETUP. 95放到磁盘上，可见它们不是只读的；那么唯一的原因是写保护。用户有机会取出磁盘并重试：这样将导致递归调用 Rename()。(这就是我们必须在模块定义文件中把各段标记为 PRELOAD 的原因：Windows 不必关心 SETUP. EXE 所在的磁盘、或是取出和重新插入磁盘。)

还有一个函数：

```
static BOOL near pascal Is Windows95(void)
    {
    return(HIBYTE(LOWORD(GetVersion())) ==95);
    }
```

GetVersion()返回31(对于 Windows3. 1和 Windows for Workgroups 3. 11)或95(对于 Windows 95)。

为了减少 PRESETUP 所占用的磁盘空间，在编译时请选择发行版选项，并指明按文件大小优化。现在，尽管空间是优先考虑的，我们仍然希望包括进一个图标，这样在 Windows 95的文件夹(它包括了所列出的各个文件的图标)当中安装程序看起来会更友好。我借用了原来的 SETUP. EXE 的图标，但变换了边框的颜色以示区别。

PRESETUP 必须编译为16位的，以便它在 Win31和 Win95下都可以运行。为此使用 Borland C++或 Microsoft Visual C++1. 5x 都可以，我选用了后者，得到的文件大小为4. 45KB，我认为这可以接受。

下一步工作是把 PRESETUP 的可执行文件 SETUP. EXE 拷贝到某个可运行它的地方。但我们首先要创建好 “genie”。

11. 6 建造 Generic Genie

Windows 程序员肯定都见过 genie(精灵)——只是它们通常被叫做 Wizard(大师)、Expert(专家)或其他抢眼的称谓。(我曾经遇到过一个共享软件，它的名字是“lizard(蜥蜴)”！真希望我也能想出这个名字！)

我们已经编写过一个 genie 程序，即 VBX Genie。它采用属性页的方式供用户访问各种可设置的选项。而更常见的情况是，这类应用程序提供给用户的是一连串的页。用户可以在页之间漫游，而所给出和所获取的信息则保证是线性序列。

这样的 genie 程序并不难编写，要做的杂事也很少。我们需要一个主表(main form)和一个 About 表(About form)。主表上至少有两个页面(frame)和一些按钮。图11. 1是设计模式下的主表。

在标记为“Finish”的页面后面还有一个您不大看得见的页面，两个页面都是同样的空白。当用这个 Generic Genie 框架创建真正的 genie 程序时，这些页面(可能还有其他的)就会填入

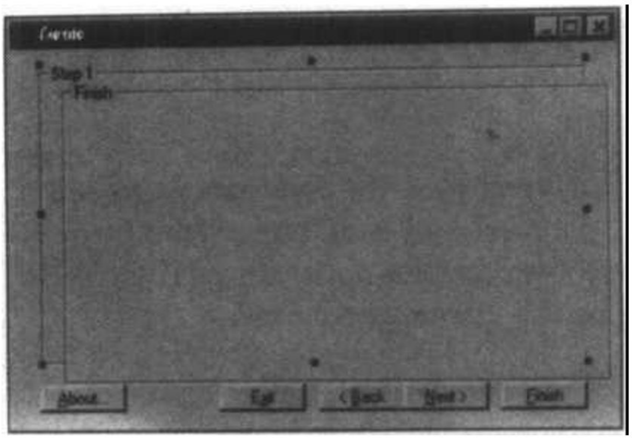

图11.1 Generic Genie的主表(设计模式下)

内容。

各个页面都是一个页面数组的元素，页面数组的各个“元素”对应一个“页面”。我倒愿意把页面的下标编号为1、2、3……这样正好与“页码”相符。然而，如果要加入新的页面，最容易的办法是，拷贝一个到剪贴板，然后粘贴；要是没有0下标，那么新页面将自动地以0为下标——而不是最后的页面号码加1。要记住这个是很烦人的，因此我决定对页码编号从0到——且慢，我差一点说成是“总页数减1”。但是，页就是页。最后的页面不能真正看作是页；仅当 Finish 按钮被单击之后它才显现出来。所以，页面的编号实际上是从0到 PageCount。

表自身命名为 Prime，我对启动表都遵循这一原则命名。下面是 Prime 表(**General**)的说明(**Declarations**)部分：

```
Opuon Explicit
Const PageCount=1
Dim CurrentPage As Integer
```

显然，当您从这个通用的 Generic genie 派生出真正的 genie 时，常量 PageCount 应作修改。当然您也可以任其为1。geme 程序只有一个数据工程页和一个“finish”页，这也是很常见的。

CurrentPage 的初始值当然为0。当装入表时，已经间接地考虑到了这一点：

```
Private Sub Form_Load()
If PageCount=1 Then
    cmd_Next. Visible=False
    cmd_Back. Visible=False
    cmd_Exit. Left=cmd_Next. Left
End if
TurnPage 1
End Sub
```

在做其他事情之前，Form_Load 子过程(**sub**)检查 genie 是否只有一页。如果真是这样，就把 Next 按钮和 Back 按钮设为不可见，并把 Exit 按钮移到 Next 按钮上面以获得最佳视觉效果。然后，调用 TurnPage 子例程，把当前页设置为第一页：

```
Private Sub TurnPage(ByVal NextPage As Integer)
```

```
if CurrentPage>0 Then
    Framel(CurrentPage-1). Visible=False
End if
CurrentPage=NextPage
Framel(CurrentPage-1). Top=150
Framel(CurrentPage-1). Left=300
Framel(CurrentPage-1), Visible=True
cmd_Back. Enabled=(CurrentPage>1)
cmd Next. Enabled=(CurrentPage<PageCount)
cmd_Finish. Enabled=(CurrentPage=PageCount)
cmd_Exit Enabled=(CurrentPage<=PageCount)
On Error Resume Next
if cmd_Finish. Enabled Then cmd_Finish. SetFocus
Refresh
End Sub
```

注意 CurrentPage 都是从1开始编号的真实页码，而页面是从0开始编号的。因此如果要用 CurrentPage 指定某个页面，应减去1。

在 TurnPage 的开始，如果先前显示了页面，就把它设为不可见的。然后，把 CurrentPage 设置为想要的页码，并设这一页对应的页面为可见。它还调整页面的位置，这样您就不必费心在设计模式下把所有的页面都层层堆叠起来了。如此一来，除开正在处理的页面，其他的都可以置之不理。

Back、Next、Finish 三个按钮是根据页面的性质来激活的。如果页还没有翻完(即，CurrentPage 小于 PageCount)，那么 Next 按钮是激活的。如果 CurrentPage 大于1，那么 Back 按钮是激活的。如果 CurrentPage 大于 PageCount，这只可能是单击了 Finish 按钮。如果是最后一页，将激活 Finish 按钮，并把输入焦点设置给它。(如果表还没有显示(本子例程首次调用就是这种情况)，试图设置焦点会导致一个错误。然而，这个错误我们可以放心地忽略；因此才有随后的 On Error 语句。)最后，调用表 的 Refresh 方法，这将确保所有的一切，包括按钮，都重新绘制。

TurnPage 做了大部分工作，因此处理 Next 按钮就轻松了，只需调用 TurnPage 即可：

```
Private Sub cmd_Next_Click()
TurnPage CurrentPage+1
End Sub
```

当 CurrentPage 与 PageCount 相等时，TumPage 使 Next 按钮失效，因此 CurrentPage 不会有越界的危险。

如您所料，Back 按钮与 Next 按钮的方法如出一辙：

```
Private Sub cmd_Back_Click()
TurnPage CurrentPage-1
End Sub
```

最终需要增强 Finish 按钮方法的功能来完成一些工作；而目前，它只是显示出最后一个

页面并显示一个毫无特色的消息框：

```
Private Sub cmd_Finish_Click()
Dim W As WaitCursor
Set W=New WaitCursor
TurnPage PageCount+1

'To do: Put your code here

MsgBox" What you wanted to do has been completed", vbInformation
Unload Me
End Sub
```

还有两个按钮分别是 About 和 Exit。Exit 按钮只是简单地下载主表：

```
Private Sub cmd_Exit_Click()
Unload Me
End Sub
```

按动 About 按钮只是简单地装入 About 表：

```
Private Sub cmd_About_Click()
About Image1. Picture=Icon
About. Show vbModal
End Sub
```

为什么我们要在这里把图标放到 About 表上呢?毕竟 About 对话框传统上都是硬编码实现的。然而在这里，我们并不遵循传统做法!这样，我们又有了另一个可重用的部件……

11.7 加入可重用的 About 对话框

在 VB3中，App 对象只有少量信息是可以被应用程序访问的。而在 VB4中，这些信息的数量已经相当多，因而实现自动的 About 对话框成为可能。

在设计期间，表外观如图11.2所示。

虚线框是 Image 控件所在的位置，图标将被放置在这里。如果调用者不在这里设置图标，Image 控件将是透明的，这样也没有什么不妥。“App Title(应用标题)”、“Version(版本)”和“Copyright(版权信息)”是显示这些信息的 Label 控件的位置。对于赠用软件，可以是一些放弃版权的文字(比如本书中的应用程序)；如果您编写的是商业软件，您也许想要把这些文字改得更具威慑力。

这三个 Label 的值是在表的 Load 事件处理程序中设置的：

```
Private Sub Form_Load()
lbl AppName=App. Title
lbl_Version—" Version" +_
```

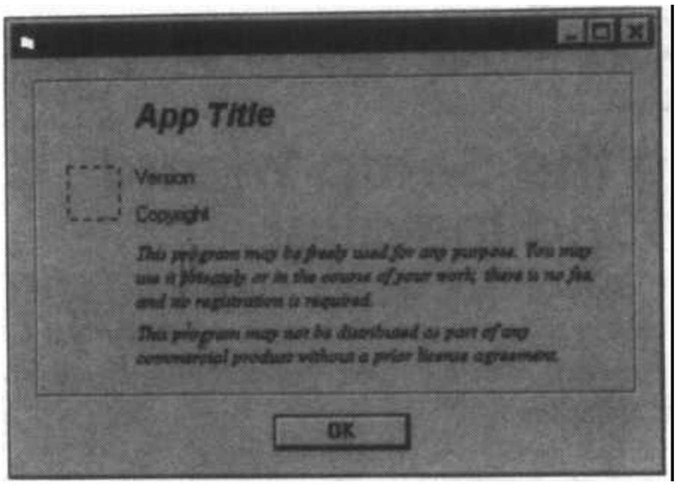

图11.2　创建 About 对话框表

```
    Format $ (App. Major)+"." +_
    Format $ (App. Minor)+"." +_
    Format $ (App. Revision)
lbl_Copyright=App. LegalCopyright
Image1. Picture=Prime. Icon
End Sub
```

如果从 Generic Genie 中运行，About 框的外观如图11.3所示。

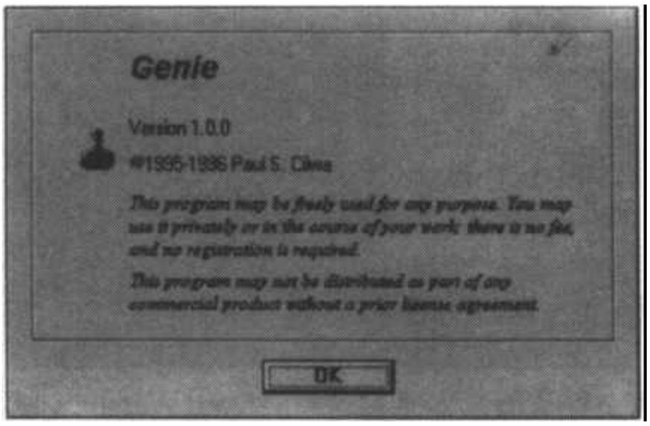

图11.3　显示运行时的 About

但请记住，对于任何用到 About 框的应用程序，它都会有相应的值。这使得 About 框成为一个有价值的可重用部件。

11.8　创建 Setup Wizard Workaround Installer

既然我们已经有了 Generic Genie，我们就有了一个可用于实际建立 Setup Wizard Workaround Installer 的框架结构。

程序的基础是 Generic Genie。在 Prime 表。我们额外再加入一个页面，这样总共就有了三个页面，还有两个控件：一个是 IniDate 控件以提供对 SETUPWIZ. INI 的访问，一个是不可见的 Label 控件以提供与 Windows 95任务栏(Task Bar)的 DDE 通信。

开始两页只是文本：第一页陈述问题，第二页列出 Workaround Installer 为解决问题所采取的步骤纲要。这些信息在上文已作过描述，所以不再赘述。

当修改完成之后，显示最后一页；由于要作的修改有很多，所以使用了进度条。运行时的最后一页如图11.4所示。

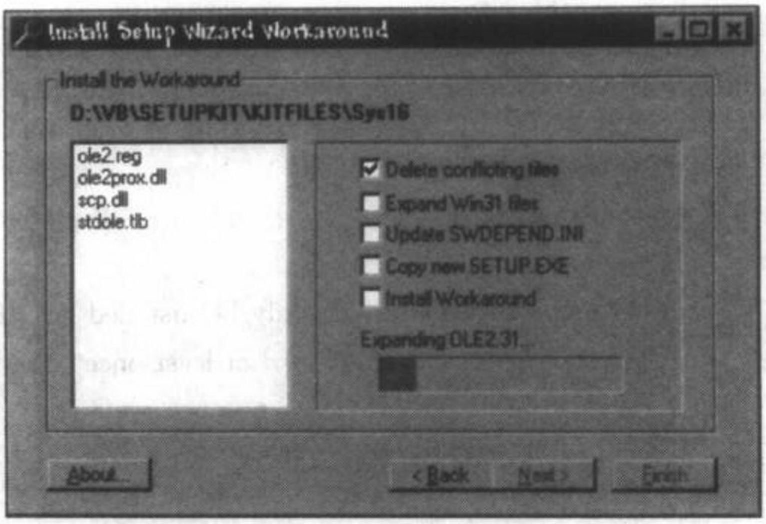

图11.4 Setup Wizard Workaround 的最后一个表

图中详细列出了主要五个步骤的细节。这些步骤反映在 cmd_Finish_Click 的处理程序中：

```
Private Sub cmd_Finish_Click()
Dim W As WaitCursor
Set W=New WaitCursor
TurnPage PageCount+1
DeleteConflictingFiles
Expand Win 31 Files
UpdateSWDepend
ReplaceSetupApp
InstallWorkaroundGenie
MsgBox" The Setup Wizard Workaround has been installed",
    vbInforrmation
Unload Me
End Sub
```

所以，要理解 Setup Wizard Workaround 安装时的工作原理，只需研究执行这五个步骤的五个子过程即可。

第一步，从 SETUPKIT\SYS16文件夹中删除掉所有可能与 Windows 3.1发生冲突的文件。但是这个文件夹在哪里?并不是每个人都把 VB 安装在 C：\VB。比如，我就把所有的开发工具都放在 D：盘。这些信息是在表的 Load 事件处理程序中获取和保存的：

```
Private Sub Form_Load()
cmd_Next.Visible=(PageCount>1)
cmd_Back.Visible=(PageCount>1)
```

```
If Not cmd_Next. Visible Then
    cmd_Exit. Left=cmd_Next. Left
End If
TurnPage 1

IniDatal. FileName=" SwDepend. ini"
IniDatal. Section=" SetupWiz"
IniDatal. KeyName=" Bootstrap"

If IniDatal. Text="" Then
    MsgBox" Setup Wizard Workaround can only be installed AFTER you have installed
Visual Basic 4. 0(16-bit)AND started the Setup Wizard at least once", vbExclamation
    End
End If

Dim i As Integer
i=InStr(IniDatal. Text,"\SETUP. EXE")
If i=0Then i=InStr(IniDatal. Text,"\SETUPA. EXE")

WizardPath=Left $ (IniDatal. Text,i-1)
lbl_KitFiles=WizardPath+" \Sys16"
lst_KitFiles. Path=lbl_KitFiles

PB. Init ProgressBar, lbl_Progress
End Sub
```

第一个代码块与 Generic Genie 中的类似。第二个代码块中，IniData 控件查询 SETUP-WIZ. INI 以获得安装工具包(setup kit)文件的位置。其路径存储在 Bootstrap 关键字下。而那些 DLL 就保存在该目录中的 SYS16子目录下。(最后一行代码与进度条有关,稍后我会讲到。)

得知 DLL 的位置之后，删除那些不想要的文件就相当容易了：

```
Private Sub DeleteConflictingFiles()
ChDrive lst_KitFiles. Path
ChDir lst_KitFiles. Path
Dirn i As Integer
For i=lst_KitFiles. ListCount To 1 Step -1
    Select Case UCase $ (lst KitFiles. List(i-1))
    Case" OLE2. REG"," OLE2PROX. DLL"," SCP. DLL"," STDOLE. TLB"
    Case Else
        SetAttr lst_KitFiles. List(i-1), 0
        Kill lst_KitFiles. List(i-1)
```

```
    End Select
Next i
lst_KitFiles.Refresh
Step(1).Value=vbChecked
End Sub
```

Select Case 语句确保除了四个 Windows 95没有的文件之外，其他所有文件都被删除掉。

下一步，把 Windows 3.11文件拷贝到该目录下以顶替被删除了的文件。这也很简单，但它花费的时间可不少，因此最好使用进度条。如此一来，我们又有了另外一个可重用的部件。

11.8.1 编写可重用的 ProgressBarClass 类的代码

VB4自己带有 Sheridan 三维控件 OCX，它是 VB3的 VBX 所对应的 OLE 控件版本。这些控件中有一个是 SSPanel。该控件的"区域填充"能力使之无愧于一个杰出的进度条。而作为程序员，您得为它的 FloodPercent 属性赋值，也就是说，您还得作些数学计算以决定进度条进行颜色填充时应该填满多少百分比。这方面的工作是重复性的工作，不如试着以某种方式把它封装起来。

VB 的用户自定义对象可以解决封装的问题。我把它叫做 ProgessBarClass。下面列出它的属性：

```
Option Explicit

Private UI As SSPanel
Private LabelUI As Control
Private Steps_As Long
Public StepsType As String

Private CurrentStep As Long
```

与在 VBX Genie 中一样，UI 是与对象相联系的控件。LabelUI 是一个附加的控件，用于显示解释性文字。如您所见，这是个可选功能。私有属性 Steps_用来存储公有属性 Steps，这是 Property Let 函数实现的。StepsType 是要发送给 LabelUI 的文本，如果提供了文本的话；CurrentStep 则在内部管理。

函数 Init 给上述属性中的两个赋了值：

```
Public Sub Init(aUI As SSPanel,aLabelUI As Control)
Set UI=aUI
Set LabelUI=aLabelUI
End Sub
```

Steps 属性通知本对象，要跟踪的进度有多少步。除了把步骤数量值赋给 Steps_属性外，还重置了其他属性，包括进度条自身：

```
Public Property Let Steps(ByVal New Value As Integer)
```

```
UI.FloodPercent=0
UI.Refresh
LabelUI.Caption=""
LabelUI.Refresh
Steps_=New Value
StepsType=""
End Property
```

应用程序可以把当前步骤序号直接设置为任意值(在可能范围之内,即从1到总步数):

```
Public Property Let Step(By Val New Value As Integer)
If New Value<1 Then New Value=1
If New Value>Steps_Then New Value=Steps_
UI.FloodPercent=((New Value-1)*100) /Steps_
UI.Refresh
CurrentStep=New Value
End ProPerty
```

SSPanel 被更新,以表明 NewValue 之前的那一步工作已经完成;然后 CurrentStep 被设置为新值。

是可以这样做,然而这个对象最常见的用法却并不是这样。别忘了,当设置 Steps 的数值时,CurrentStep 被初始化为0。而处理过程通常是从第一步开始的,并且一直继续下去;因此在每步的开始,都要调用 StepStarted 方法:

```
Public Sub StepStarted(Optional StepName)
UI.FloodPercent=(CurrentStep *100)/Steps_
UI.Refresh
CurrentStep=CurrentStep+1
If CurrentStep>Steps_Then CurrentStep=Steps_
If Not LabelUI Is Nothing Then
    If IsMissing(StepName)Then StepName=""
    Dim Caption As String
    Caption=Trim $ (StepsType+" "+StepName)
    If Caption>"" Then Caption=Caption+" …"
    LabelUI.Caption=Caption
    LabelUI.Refresh
End If
End Sub
```

如果在对象初始化时给出了 LabelUI,该任选参数就为本步骤提供了一个标题。它同 StepsType 参数联合使用。常见的联合是把 StepsType 设置为“Copying(正在拷贝)”,并用当前正在拷贝的文件名作为参数调用 StepStarted。

本步结束之后，调用下面的方法：

```
Public Sub StepCompleted()
UI.FloodPercent=(CurrentStep 100)/Steps
UI.Refresh
If Not LabelUI is Nothing Then
      LabelUI.Caption=""
LabelUI.Refresh
End if
End Sub
```

有时，完成一个操作所用的步骤会比预料的要少。在这种情况下，可以调用 StepsCompleted 方法作清理工作：

```
Public Sub StepsCompleted()
UI.FloodPercent=100
UI.Refresh
CurrentStep=Steps
If Not LabelUI Is Nothing Then
      LabelUI.Caption=""
      LabelUI.Refresh
End If
End Sub
```

但有一点是肯定的，即删除 ProgressBarClass 对象时，将激活 Terminate 函数：

```
Private Sub Class Terrninate()
If Not UI Is Nothing Then
      UI.FloodPercent=0
End If
End Sub
```

把进度条的 FloodPercent 设置为0，它将恢复到初始外观。

现在，您知道 Prime 表的 Load 事件处理程序中最后一行代码的意思了：此时正在初始化一个 ProgressBarClass 类的对象 PB，以给出进度条和它上面的标签。

11.8.2 展开 Win31文件

我们没有必要在第一步里使用进度条，因为删除那些文件所花费的时间极短，没有什么值得担心。然而当有13个文件需要展开时，花费的时间就长了，有必要显示工作进度。

展开过程需要调用 Expand 函数13次，每展开一个文件都调用一次：

```
Private Sub ExpandWin31Files()
PB.Steps=17
PB.StepsType=" Expanding"
```

```
    Expand MakePath ( "", App, Path," Compobj "," .DL_ "), lbl_ KitFiles + " \
COMPOBJ. DLL"
    Expand MakePath ( "", App, Path," Ctl3Dv2 "," .DL_ "), lbl_ KitFiles + " \
CTL3DV2. DLL"
  Expand MakePath("",App,Path,"Ole2",". 31_"), lbl_KitFiles+" \OLE2. 31
  Expand MakePath("",App,Path,"Ole2",". DL_"), lbl_KitFiles+" \Ole2. DLL"
    Expand MakePath ( "", App, Path," Ole2Conv "," .DL_ "), lbl_ KitFiles + " \
OLE2CONV. DLL"
    Expand MakePath ( "", App, Path," Ole2Disp "," .DL_ "), lbl_ KitFiles + " \
OLE2DISP. DLL"
    Expand MakePath ( "", App, Path," Ole2Nls "," .DL_ "), lbl_ KitFiles + " \
OLE2NLS. DLL"
    Expand MakePath ( "", App, Path," Ole2Prox "," .31_ "), lbl_ KitFiles + " \
OLE2PROX. 31"
  Expand MakePath("",App,Path,"Scp",". 31_"), lbl_KitFiles+" \SCP. 31"
  Expand MakePath("",App,Path,"StdOle",". 31_"), lbl_KitFiles+" \STDOLE. 31"
    Expand MakePath ( "", App, Path," Storage "," .DL_ "), lbl_ KitFiles + " \
STORAGE. DLL"
    Expand MakePath ( "", App, Path," TypeLib "," .DL_ "), lbl_ KitFiles + " \
TYPELIB. DLL"
  Expand MakePath("",App,Path,"VShare",". 38_"), lbl_KitFiles+" \VSHARE. 386"
  Expand MakePath("",App,Path,"Ole2",". DL_"), lbl_KitFiles+" \VSHARE. 386"
    Expand MakePath ( "", App, Path," Ole2Prox "," .DL_ "), lbl_ KitFiles + " \
VSHARE. 386"
  Expand MakePath("",App,Path,"SCP",". DL_"), lbl_KitFiles+" \VSHARE. 386"
  Expand MakePath("",App,Path,"StdOle",". TL_"), lbl_KitFiles+" \VSHARE. 386"
  PB. StepsCompleted
  lst_KitFiles. Refresh

  Step(2). Value=vbChecked
  End Sub
```

函数 Expand 进行实际的展开工作：

```
Private Sub Expand(By Val SourceFile As String,DestFile As String)
Dim Drive As String, Dir As String, Root As String, Ext As String
SplitPath DestFile, Drive, Dir, Root, Ext
PB. StepStarted Root+Ext
CopyCompressedFile SourceFile, DestFile
End Sub
```

请注意，如果一步的结束和下一步的开始之间没有多少时间间隔，就没有必要调用 ProgressBarClass 对象的 StepCompleted 方法。在这里不会有这种情况。

FILES. BAS 模块中的函数 CopyCompressedFile 进行实际的拷贝工作：您可能会因 VBX-Genie 而记得它。为帮助您回忆，这里我给出了它的代码：

```
Public Sub CopyCompressedFile( _
    By Val SourceFilename As String, _
    By Val DestFilename As String)
Dim Source As OFSTRUCT, SourceFile As Integer
Dim Dest As OFSTRUCT, DestFile As Integer
SourceFile=LZOpenFile(SourceFilename,
    Source,OF_READ)
DestFile=LZOpenFile(DestFilename,_
    Dest,OF_CREATE)
LZCopy SourceFile, DestFile
LZClose SourceFile
LZClose DestFile
End Sub
```

LZOpenFile()、LZCopy()、LZClose()当然都 Windows API 函数。它们拷贝由 COMPRESS. EXE 程序压缩过的文件，并把它展开到所需的位置。

现在的问题是，这些文件从哪里来?本书附带的 CD-ROM 上有经 Microsoft 公司授权的、并与 Win31兼容的最新版本的文件。每个文件都是用类似于下面的命令行压缩得到的：

```
compress-r OLE2. DLL
```

"-r"标志告诉压缩程序，把未被压缩的文件名包括到压缩后的文件里。我们是这样做的，但是 LZCopyFile()并没有利用这个信息，因为即使没有这信息，它也这样做。(另外有一些函数确实需要用到这信息。)

大多数文件压缩后以它们自己的名字存储。您也许注意到了那些扩展名为. 31的文件，它们会与相应的 Win95文件发生冲突，而且它们不是 Windows95自带的文件，所以还得由我们提供。通过把扩展名改为. 31，我们避免了命名冲突。

11. 9 更新 SWDEPEND. INI

还记得吧，SWDEPEND. INI 文件被 Setup Wizard 用来记录哪些文件需要别的文件。我们得做一些改动，为此专门单独安排了一个步骤。我们来一段一段地读代码吧。

在第二步里，我们利用进度条来监控 OLE DLL 的展开情况。第二步的工作完成之后，我们又可以用它来跟踪后面的步骤：

```
Private Sub UpdateSWDepend()
PB. Steps=5
```

```
PB.Step=2
PB.StepStarted
```

现在我们要修正 SWDEPEND INI 文件。第一项任务是告诉它把自举文件由 SETUP.EXE 替换为 SETUPA.EXE：

```
IniData1.FileName=" SWDEPEND.INI"
IniData1.Section=" SetupWiz"
IniData1.KeyName=" Bootstrap"
IniData1.Text=WizardPath+" \SETUPA.EXE"
```

请记住原有的 SETUP.EXE 文件不久就会被改名为 SETUPA.EXE，所以在这里我们其实并没有改变什么。但是，我们仍然希望把预安装程序(即 SETUP.EXE)包括进去。在 SWDEPEND INI 文件中再加入一个关键字 Uses*n* 即可做到这一点。"n"必须是一个新值，正好比现有的最大值大1。下面的代码计算出这个值，并加入新项：

```
Dim i As Integer
Do
    i=i+1
    IniData1.KeyName=" Uses" +Format $ (i)
Loop Until IniData1.Text=""
IniData1 Text=lbl_KitFiles+ "\SETUP.EXE; 0"
```

预安装程序不要被 Setup Wizard 压缩，这一点很重要——否则，它怎么运行呢?它是用来解压缩其他文件的!文件名后面加上的"；0"就是要通知 Setup Wizard 不要压缩本文件。

下一项任务是修正 Visual Basic Setup 安装程序所生成的 SWDEPEND.INI 文件中的一个拼写错误：

```
IniData1.Section=" CRYSTL16.OCX"
IniData1.KeyName=" Uses2"
IniData1.Text=" CRXLATE.DLL"
```

最后，我们得加上对另外那四个文件的引用：

```
IniData1.Section=" OLE2.DLL"
i=0
Do
    i=i+1
    IniData1.KeyName=" Uses" +Format $ (i)
Loop Until IniData1.Text=""
IniData1.Text=" OLE2PROX.31"
IniData1.KeyName=" Uses" +Format $ (i+1)
IniData1.Text=" OLE2.31"
IniData1.KeyName=" Uses" +Format $ (i+2)
```

```
IniDatal. Text=" SCP. 31"
IniDatal. KeyName=" Uses" +Format $ (i+3)
IniDatal. Text=" STDOLE. 31"

Step(3). Value=vbChecked
End Sub
```

11.10 安装 PRESETUP

预安装程序已命名为SETUP. EXE。为了避免混淆，而且还由于这是个小型程序，所以我决定把它作为 Setup Wizard Workaround Installer 的一个资源。这样得编写一个 INSTALL. RC 资源文件，指定文件，并为这资源命名：

SETUP EXE   PRESETUP\SETUP. EXE

就象 VBX Genie 那样，必须使用16位的资源编辑器(比如 Microsoft 公司的 App Studio 或 Borland 公司的 Resource Workshop)以产生所需的 INSTALL. RES 文件。

这些完成之后，原始 SETUP. EXE 改名以及新程序“即位”的代码就一目了然了：

```
Private Sub ReplaceSetupApp()
PB. StepStarted
ChDrive lbl_KitFiles
ChDir lbl KitFiles
GenerateBinaryFile "SETUP"" ExE", "Setup. exe"
ChDir ".."
On Error Resume Next
Name " setup. exe" As" SetupA. exe"
Step(4). value=vbChecked
End sub
```

11.11 安装 Workaround

最后一步：安装 workaround 自身—用户每次运行 Setup Wizard 都要运行 Workaround 程序。

现在的问题是，我们还没有编写 Setup Wizard Workaround. 或者说，你还没有，而我已经编写了 Workaround；我是在编写安装程序之前完成 Workaround 的编写工作的。但是我知道，如果以这种次序解释我的代码的话，不容易讲清楚。所以，我们假设已经有了一个名为 SWWRKRND. EXE 的文件。假设由于天意的巧合，这个可执行文件偶然被我们拾到，于是我们就可以马上安装它，而实际上我们要等到下一节才会“编写”这个程序。

作了如上的假设后，下面就是它的安装过程：

```
Private Sub InstallWorkaroundGenie()
PB. StepStarted
  Expand MakePath ( "", App. Path," SWWrkrnd "," EX_ "), WizardPath + " \
```

```
SWWRKRND.EXE"

    Dim Try As Integer
    On Error Resume Next
    For Try=1 To 20
        lbl_Progman.LinkTopic=" PROGMAN|PROGMAN"
        If Err=0 Then
          Exit For
        End if
        DoEvents
    Next Try
    lbl_Progman.LinkMode=2
    For Try=1 To 10
        DoEvents
    Next.Try
    lbl_Progman.LinkTimeout=100

    lbl_Progman.LinkExecute" [CreateGroup(""Visual Basic 4.0(16-bit)"", 1)]"
    lbl_Progman.LinkExecute" [ReplaceItem(Application Setup Wizard)]"
    Err=0
    lbl_Progman.LinkExecute" [AddItem(" + WizardPath + "\SWWRKRND.EXE,Applica
tion setup Wizard,,,)]"
    PB.StepCompleted
    Step(5).Value=vbChecked
    End Sub
```

第一部分的代码很简单，展开压缩文件并归位。剩下的代码利用DDE确保任务栏(它仍然管自已叫程序管理器(Progman))把已有的Application Setup Wizard图标替换成Setup Wizard Workaround的图标。

11.12 关于Setup Wizard Workaround程序

这是我们最后要讨论的程序，它修改Setup Wizard产生的分布盘文件，使它们与Win31和Win95能够和平共处。

一般来说，Workaround都是与Setup Wizard配合使用的，那么为什么不让Workaround自动Setup Wizard呢?可以把Setup Wizard的图标从Start菜单替换掉；然后当您单击它时，Setup Wizard将自动启动，而当Setup Wizard的戏唱完之后，Workaround再登台。

这将如何实现?我们来看看Workaround的代码，这代码自然是以Prime表为舞台的。下面是Load事件的处理程序：

```
Private Sub Form_Load()
```

```
IniDatal. FileName="SwDepend. INI"
IniDatal. Section="SetupWiz"
IniDatal. KeyName="Bootstrap"
Dim Drive As String,Path As String,Root As String,Exi As String
SplitPath IniDatal. Text,Drive,Path,Root,Ext
KitFilesPath=Drive+Path+"\Sys16"

ChDrive Drive
ChDir Path

Dim TasKID As Integer,Dummy As Integer
TaskiD-Shell("SetupWiz. exe",vbNonnalFocus)
Do While GetModuleUsage(TaskID)>0
     Dummy=DoEvents ()
Loop

cmd Next. Visible=(PageCount>1)
cmd Back. Visible=(PageCount>1)
if Nof cmd_Next. Visible Then
     cmd_Exit. Left=-cmd_Next.Left

End if
TurnPage 1
End Sub
```

第一段代码很眼熟，它直接来自 Generic Genie。但是在表中，我还加入了一个 IniData 控件，它可提供对 Setup Wizard 的 INI 文件的访问。通过检查 SetupWiz 节的 Bootstrap 关键字的值，可以找到 Setup Wizard 所在位置的路径。在路径的后面续上“\SYS16”，我们就得到了 OLEDLL 所在的位置(这个值我们将保存起来以供后用。)然后把当前驱动器和目录改换到那个路径。

接下来，我们运行 Setup Wizard。VB 的 Shell 函数返回一个“任务标识(Task ID)”：这个值可以传递给 Windows API 函数 GetModuleUsage ()，它返回该任务有多少副本正在运行，在副本数目降至0——即是 Setup Wizard 被关闭——之前，我们一直都会在 DoEvents () 里循环。此后，Workaround 就接过工作，而对 Load 事件的处理也告一段落。

顺便插一句，在 VB4 中，API 的 Declare 语句不能放到表的(General)的说明(Declarations)部分，这是 VB4规定比 VB3更严格的几个限制之一。Delcare 语句也不能进入类模块，它们只能放在代码模块当中。因此我已经养成了这个习惯：在多数工程里都包含一个 WINAPI. BAS 模块，其中有工程中用到的 API 函数的 Declare 声明语句。比如 Workaround 只需要一个 Declare 语句：

```
Option Explicit
```

```
Declare Function GetModuleUsage_
    Lib "Kernel"_
    (ByVal hModule As Integer)As Integer
```

这样，如果用户不是试图在 Setup Wizard 运行的时候去看它的图标，那么看到的将是 Workaround 的第一页，如图11-5所示。

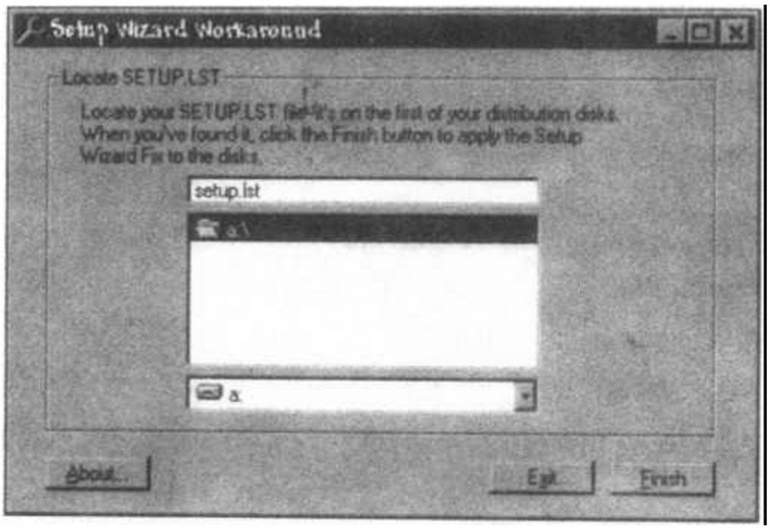

图11.5 Setup Wizard Workaround 对话框的第一页

这一页只是 Directory、Drive 和 File 控件的一般性应用。开始时 Drive 控件寻找当前驱动器；Directory 控件寻找该驱动器的当前目录。如果用户用 Drive 控件改换了活动的驱动器，新的驱动器值将被发送给 Directory：

```
Private Sub Drive1_Change()
On Error Resume Next
Dir1. Path=Drive1. Drive
ChDrive Drive1. Drive
End Sub
```

此处的 On Error 语句是为了防止出现系统崩溃，造成崩溃的罪魁祸首显然是一个 Windows95错误，我是在做这个工程期间与它不期而遇的；有时，某个状态相当良好的驱动器会莫名其妙地不可访问，只得关机重新启动。(当然,这种情况不常发生。)

如果 Directory 改变了——可以是用户直接改变了目录或者是间接改变了活动驱动器——File 控件都将被更新：

```
Private Sub Dir1_Change()
File1. Path=Dir1. Path
ChDir Dir1. Path
End Sub
```

File 控件在最初(在设计时)是指定 "SETUP. LST" 作为 "通配符"，因此如果驱动器和目录中不含有 SETUP. LST 名字的文件,File 都显示为空的。由于在一个目录下只可能有一个这样名字的文件，所以 File 控件—它其实是一个列表框—被缩小了，看起来它非常类似一个编辑(edit)控件或者是标签(label)控件。

在找到 SETUP. LST 之前，我们不希望 Finish 按钮被激活：

```
Private Sub File1_PathChange()
cmd_Finish. Enabled=(File1. ListCount>0)
End Sub
```

还记得吧，在 Generic Genic 中，当显示最后一页时，Finish 按钮被激活。在这里我们得作一点变通：

```
Private Sub TumPage(ByVal NextPage As Integer)
if CurrentPage>0 Then
    Frame1(CurrentPage-1). Visible=False
End if
CurrentPage=NextPage
Frame1(CurrentPage-1). Top=150
Frame1(CurrentPage-1). Left=300
Frame1(CurrentPage-1). Visible=True
cmd_Back. Enabled=(CurrentPage>1)
cmd_Next. Enabled=(CurrentPage<PageCount)
cmd_Exit. Enabled=(CurrentPage<=PageCount)
cmd_Finish. Enabled=
    (CurrentPage=PageCount)And
    (File1. ListCount>0)
On Error Resume Next
if cmd_Finish. Enabled Then cmd_Finish. SetFocus
Refresh
End Sub
```

即使是在 Workaround 这个具体实现的代码里，PageCount 也还是1，在程序中我保留了那段测试代码，以备在必要时加入其他页。

用户在找到 SETUP. LST 之后，就可单击 Finish 按钮：

```
Private Sub cmd_Finish_Click()
Dim W As WaitCursor
Set W=New WaitCursor
TumPage PageCount+1
Finish
MsgBox "Your distribution disks have been modifed" +_
    "for safe use on Win31 and Win95 systems", _
    vbInformation
Unload Me
End Sub
```

要做的事情太多，不能都一股脑都堆在这里；于是我分离出一个名为 Finish 的子例程。但是为了更好地理解即将展现在我们面前的代码，需要先看一些其他部件。

第一个部件是一些集合。这都定义在 Prime 的(General)(Declarations)部分：

```
Dim BootStrap As New Collection
Dim Files As New Collection
Dim Setup As New Collection
```

这些集合分别代表了 SETUP.LST 文件当中的三节。BootStrap 节和 Files 节中的各项都是用户自定义类 FileEntry 的对象。我们来看看该类的属性，都定义在 FILE. CLS 的(General)(Declarations)部分：

```
Option Explicit
Private DiskNo As Integer
Private Split As Boolean
Public FileName As String
Public TargetName As String
Private TargetPath As String
Private Register As String
Private Remote As String
Private FileDate As Date
Private FileSize As Long
Private Version As String

Public Enabled As Boolean

Private SplitText(True To False)As String
```

据 Visual Basic 的文档，SETUP. LST 中各项都由这些字段构成。对某项来说，并不是所有的字段都要用到；但我们仍然应该为它留出这种可能性。Enabled 属性是个显著的例外；我加入它的目的是为了在创建新的 SETUP. 31和 SETUP. 95两个文件时可以分别选用或不用某些项。

SplitText 数组是一个内部变量，用于处理 Microsoft 在 SETUP. LST 中所用到的常量。在创建类对象时，数组被初始化：

```
Private Sub Class_Initialize ()
SplitText(True)= "SPLIT"
End Sub
```

每创建一项都要用到 Init 函数，以设定该项的那些属性：

```
Public Sub Init(ByVal aDiskNo As String,_
    ByVal aSplit As String,_
    ByVal aFilename As String,_
    ByVal aTargetname As String,_
    ByVal aPathname As String,_
    ByVal aRegistel As String,_
    ByVal aRemote As String,_
```

```
    ByVal aDate As String, _
    ByVal aSize As String, _
    ByVal aVersion As String)
DiskNo=Val(aDiskNo)
Split=(aSplit="SPLIT")
FileName=aFilename
TargetName=aTargetname
TargetPath=aPathname
Register=aRegister
Remote=aRemote
if aDate>""Then FileDate=CDate(aDate)
FileSize=Val(aSize)
Version=aVersion
Enabled=True
End Sub
```

项就是这样被“制造”出来。类还应能够重建初始的文本。这由一个 Property Get 函数完成：

```
Public Property Get Text()
Dim Result As String
Result=Format $ (DiskNo)+ "," +SplitText(Split)+ "," +_
    FileName+ "," +TargetName+ "," +TargetPath+ "," +_
    Register+ "," +Remote+ "," +_
    Display(FileDate)+ "," +_
    Display(FileSize)+ "," +Version+ ","
While Right $ (Result,1)= ","
    Result=Left $ (Result,Len(Result)-1)
Wend
Text=Result
End Property
```

各字段只是简单地连接起来，以逗号分隔。有的字段在内部以二进制方式存储；函数 Display 巧妙地完成了对它们的格式转换，这要归功于函数的一个 Variant 数据类型参数：

```
Private Function Display(Data As Variant)As String
If TypeName(Data)= "Date" Then
    If Data<CDate("1/1/1980")Then
        Display= ""
    Else
        Display=FormatS(Data,"m/d/yyyy")
    End If
Else
    If Data=0Then
        Display= ""
```

```
        Else
            Display=Format $ (Data)
        End If
    End If
    End Function
```

您已经知道了 FileEntry 类是如何实现的，现在来看看它的运用情况。子例程 LoadSection 读取 SETUP. LST 给定某节的所有项，为每一项创建一个 FileEntry 对象，并把 FileEntry 对象加入指定集合：

```
    Private Sub LoadSection(Section As String, List As Collection)
    IniDatal Section=Section
    Dim File As FileEntry
    Dim i As Integer
    For i=1 To IniDatal. KeyCount
        iniData1. KeyName=iniData1. KeyList(i-1)
        If IniDatal. Text> "" Then
            Set File=New FileEntry
            File. Init IniData1. List(0),_
                IniData1. List(1),_
                IniData1. List(2),_
                IniData1. List(3),_
                IniData1. List(4),_
                IniData1. List(5),_
                IniData1. List(6),_
                IniData1. List(7),_
                IniData1. List(8),_
                IniData1. List(9)
            List. Add File. IniData1. List(2)
        End If
    Next 1
    End Sub
```

用于刻画文件的那十项之间是以逗号间隔的；所以把 IniData 控件的 ListDelimiter 属性设置为 “,” 以便它能够识别与分离。第三项是实际的文件名，检索集合元素时就要把它作为关键字。(文件名是传递给集合的 Add 方法的第二个参数。)

在一个 SETUP. LST 文件里总共有三节，其中的两节全部是文件项；因此上面的子例程可用于它们两个。第三节中是几个必须存储的简单文本项。SETUP. CLS 中定义的 SetupClass 类就是为处理第三小节的内容而设计的。该类的对象有两个简单属性：

```
    Option Explicit

    Public Key As String
```

```
Public Value As String
```

两个属性都是公有的，而 Init 过程可以更方便地对两者进行初始化：

```
Public Sub Init(ByVal aKey As String,ByVal aValue As String)
Key=aKey
Value=aValue
End Sub
```

类 SetupClass 是可重用的。举例来说，该类的对象与一个标准集合相结合，就可得到一本“字典”，即按关键字排序的元素集合。回到 PRIME. FRM，现在我们要在子例程 LoadSetupSection 中使用 SetupClass 类：

```
Private Sub LoadSetupSection()
IniData1. Section= "Setup"
Dim i As Integer
Dim Item As SetupEntry
For i=1 To IniData1. KeyCount
    IniData1. KeyName=IniData1. KeyList(i-1)
    Set Item=New SetupEntry
    Item. Init IniData1. KeyName, IniData1. Text
    Setup. Add Item, IniData1. KeyName
Next i
End Sub
```

有了这些准备工作，我们来看 Finish 过程。它仍然相当冗长；因此我们分段加以研究。第一，重置 IniData 控件，得到用户刚找到的 SETUP. LST 文件：

```
Private Sub Finish()
Dim i As Integer
IniData1. FileName=MakePath("",File1. Path,File1. FileName,"")
```

下一步是调用 LoadSection。我们已经知道，它的作用是列举出 SETUP. LST “BootStrap” 节的各个文件，并把它们添加到 BootStrap 集合中：

```
Label1(1)= "Loading" +IniData1. FileName+ "..."
Refresh
LoadSection "BootStrap", BootStrap
CorrectBootStrap
LoadSection "Files", Files
LoadSetupSection
Kill IniData1. FileName
```

我们得把 Win31的 OLE 文件的展开名改过来。并且要去掉 SETUP. EXE 这一项，它在那儿只是起到占位的作用，而(重新)以 Workaround Installation 取而代之。然后，装入 Files 节和

Setup 节的内容。当所有 SETUP. LST 的信息都已安全地存放到内存里后，我们就可以删除这个文件，以便替换工作的开展。

一个要替换的是 SETUP. 31，它是 SETUP. LST 用于 Win31机器的版本。IniData 控件被重置，以创建这个新文件(设置第一个关键值即可)：

```
Label1(1)= "Creating SETUP. 31…"
Refresh
Dim Drive As String, Path As String, Root As String, Ext As String
SplitPath IniData1. FileName, Drive, Path, Root, Ext
IniData1. FileName=MakePath(Drive,Path,Root,". 31")
BootStrap("OLE2. RE_"). Enabled=False
BootStrap("OLE2PROXDL_"). Enabled=False
BootStrap("SCP. DL_"). Enabled=False
BootStrap("STDOLE. TL_"). Enabled=False
DumpSection "BootStrap", BootStrap
DumpSection "Files", Files
DumpSetupSection
```

集合里的元素随后由 CorrectBootStrap 过程加以修正：

```
Private Sub CorrectBootStrap ()
BootStrap("OLE2. 3"). TargetName= "OLE2. REG"
BootStrap("OLE2PROX. 3_"). TargetName= "OLE2PROX. DLL"
BootStrap("SCP. 3_"). TargetName= "SCP. DLL"
BootStrap("STDOLE. 3_"). TargetName= "STDOLE. TLB"
BootStrap. Remove "SETUP. EXE"
End Sub
```

BootStrap 集合中的那些在安装时不应该拷贝到 Win31机器的文件都设为无效。然后把三节的所有内容都"转储"到 SETUP. 31，即都写入 SETUP. 31。(稍后我们会讲到完成这转储工作的代码。它很简单。)

创建 SETUP. 95与此类似……这样，所有的工作就完成了：

```
Label1(1)= "Creating SETUP. 95…"
Refresh
IniData1. FileName=MakePath(Drive,Path, Root,". 95")
BootStrap("OLE2. RE_"). Enabled=True
BootStrap("OLE2PROX. DL_"). Enabled=True
BootStrap("SCP. DL_"). Enabled=True
BootStrap("STDOLE. TL_"). Enabled=True
BootStrap("OLE2. 3_"). Enabled=False
BootStrap("OLE2PROX. 3_"). Enabled=False
BootStrap("SCP. 3_"). Enabled=False
```

```
BootStrap("STDOLE. 3"). Enabled=False
DumpSection "BootStrap", BootStrap
DumpSection "Files", Files
DumpSetupSection
End Sub
```

转储 BootStrap 节或者 Files 节即是遍历相应的集合，依次得到每一项，然后利用属性"Text"重建出正确字符串作为关键字值：

```
Private Function DumpSection(Section As String,List As Collection)
IniDatal. Section=Section
Dim i As Integer
For i=1 To List Count
    If List(i). Enabled Then
        IniDatal. KeyName= "File" +Format $ (i)
        IniDatal. Text=List(i). Text
    End If
Next i
End Function
```

代码的简单程度令人难以置信；可是别忘了，其实这是 FileEntry 对象的功劳。

转储 Setup 节是一件类似的任务：

```
Private Function DumpSetupSection()
IniDatal. Section= "Setup"
Dim i As Integer
For i=1 To Setup. Count
    IniDatal. KeyName=Setup(i). Key
    IniDatal. Text=Setup(i). Value
Next i
End Function
```

11.13 打点行装回家

当我本以为已经为 VBX Genie"炮制"出了一个快速安装程序时，不料却又费了额外几周时间去查明 SetupWizard 为什么不好好工作、为什么和怎样手工地进行软件设置、怎样为别的使用者减轻设置负担以及怎样把设置过程打包为可重用部件。有人抱怨说，Microsoft 公司显然没有把他们的产品放到安装了不同软件和硬件的终端用户机器设备上加以测试。我却不怨天尤人！如果不是因为他们和他们行事的方式，我就会错过这场意外的经历。您也是，您可能已草草翻过了这20几页。

难以忘怀的经历往往是在您不曾料到的时刻到来。

第十二章　用作查找和替换对话框的 VBX

12.1　对话框的窘境

Windows 最初发行时，所有的 Windows 应用程序——记事本、书写器、PIF 编辑器、终端仿真程序等等——都使用相同的 File Open 和 File Save 对话框。这样带来的是相当程度的恒定性，然而在编写我们自己的 Windows 应用程序时，我们发现，如果想要在程序中用这些对话框，就得要复制它们。Charles Petzold 在他的著作《Windows 程序设计》一书中介绍了如何做，但是这个过程仍嫌麻烦，因为我们还得重复那些已经做好的工作。

Boriand 公司在发行 Turbo Pascal for Windows 以及后来的 TurboC＋＋for Windows 时，充分考虑到了这一点：它们包含了用于打开文件和保存文件的对话框，并鼓励我们使用。诚然，在 Turbo Pascal 的打开文件对话框过程的代码中有一个小缺陷，尽管这样……

有这样一个有趣的现象，在多数公司倾向于购买 Microsoft 公司编译器的同时，多数个人用户却购买 Borland 公司的编译器。还有许多异军突起的软件并不是产自公司的摩天大楼，而是产自小车库。因而许多其貌不扬的应用软件——事实上都是共享软件——使用了 Borland 公司的打开文件和保存文件的对话框。这一点您能够很容易地辨别出来，因为这些应用程序都有一个带有绿色"OK"标记的图形按钮和一个通用的表示否定含义的"Cancel"标记的图形按钮。而 Microsoft 的对话框使用的则是更为传统的 OK 按钮和 Cancel 按钮。图12.1和12.2显示出了两者的差别。

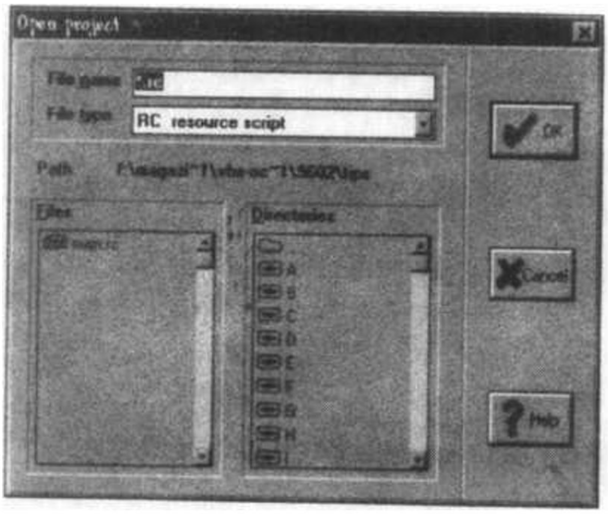

图12.1　Borland 标准 File Open 对话框

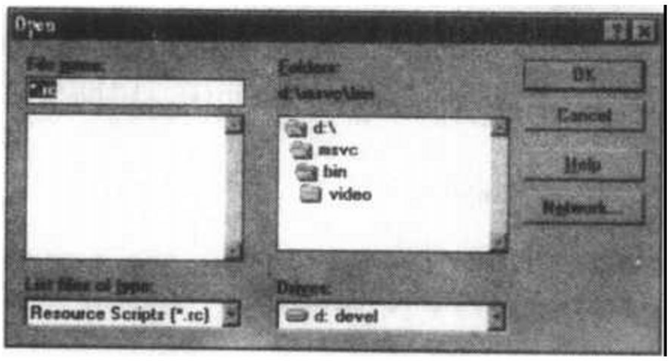

图12.2　Microsoft 标准(16位)File Open 对话框

有这样的传闻，说是 Microsoft 公司曾阻止应用程序开发商获取任何好处。Microsoft 公司辟谣之事正印证了这一点。这样，他们在应用软件市场上就拥有"不公平优势"。从 Borland 推出的对话框看来，事实显然不是这样。不管怎样，Microsoft 公司通过发行 Common Dialog DLL（通用对话框的动态连接库）解决了它所面临的问题。这个已经被 C 和 C＋＋程序员使用了几年的 DLL 提供的不仅有文件打开和保存对话框，还有颜色、字体和打印对话框。它也包括了 Find（查找）和 Relplace（替换）对话框。

从 Visual Basic 可访问除 Find 和 Replace 对话框之外的所有对话框，因为通用对话框是保

存在一个DLL中，相应的例程是作为Windows API的一部分出版(从Windows 3.1开始)。然而，Find和Replace对话框都需要父窗口处理用户消息（custom message)。然而，Find对话框可以要求用户给出要查找什么，而实际上它什么也查找不出，因为它不知道您的文档在内部的（或外部的）是怎样存储的。所以定制消息是要在适当的时候发送；而这正是您调用代码完成查找工作的时候。

很不走运，在Visual Basic中无法处理用户消息，因而Find和Replace这两种通用对话框就不能为VB程序员效力了。

在Common Dialog VBX投入市场时，我曾幻想它会包括Find和Replace对话框。VBX里有Open和Save，也有Color、Font和Print……但就是没有Find和Replace!这两个对话框，从前用不了，现在还是用不了，看来局面会持续下去，这似乎很荒谬……然而事实就摆在那里。似乎是为了补偿，Common Dialog VBX加入了一个Common Dialog DLL所不具备的功能，即可以轻松访问Windows Help的机制。但这个改进无关痛痒。

既然已经有了完整的VBX Genie，我们就不必非得搭Microsoft公司的车，我们可以自己造车；我们的"车"可以为Find和Replace对话框提供所需的回调函数。我们的这个项目可作为一个绝佳的例子，用以说明如何利用功能强大的软件部件解决一个棘手的编程问题。

12.2 Find和Replace对话框

想要封装Find和Replace对话框，最好先了解它们的工作原理。跟所有的通用对话框一样，Find和Replace对话框也需要程序员填好一个结构，然后把结构传递给一个函数，由这个函数负责对话框的实际显示。当用户关闭对话框时，结构可以更新，也可查询。

Find和Replace常常是一起工作，所以它们使用了同样的结构，而实际上如果愿意，您甚至可以只用到这个结构的一个实例。这样做的话，用户想要"find(查找)"的字符串将作为种子值出现在此后的Replace对话框中。用户喜欢这种效果，他们把这当成是一种"特色"而不是缺憾。

下面是这个结构的C语言定义：

```
typedef struct tagFINDREPLACE
 {
    DWORD      lStructSize;
    HWND      hwndOwner,
    HINSTANCE hInstance;
    DWORD      Flags;
    LPSTR     lpstrFindWhat;
    LPSTR     lpstrReplaceWith;
    UINT     wFindWhatLen;
    UINT     wReplaceWhatLen;
    LPARAM      lCustData;
    UINT     (CALLBACK *lpfnHook)(HWND,UINT,WPARAM,LPARAM);
    LPCSTR     lpTemplateName;
```

} FINDREPLACE;

字段 lStructSize 应填入本结构的大小；这将用于某个“版本”字段的场合以支持将来的扩充。hWndOwner 是要从对话框接收消息的窗口句柄；也就是说，即使 FindReplace 控件在运行时刻是不可见的，它也不可能具有 MODEL_fGraphical 风格；它将不得不视其所服务的窗口而定。

如果我们想把自己的模板用于 Find 和 Replace 对话框，就会用到 hInstance；但如果我们真这样做，在 Visual Basic 中创建对话框就可以了，不是吗?所以这里只需把 hInstance 设定为 NULL 即可。

Flags 就是作上述用途的；但 Flags 还可以派上其它用场。需要有 Flags 标志设定查找方向(“Up(向上)”或“Down(向下)”)、替换范围(“replace all(全部替换吗?)”)、是否区分大小写、查找时是否全字匹配。

字符串指针 lpstrFindWhat 和 lpstrReplaceWith，它们只是指针而已。您得给它们分配内存；您还得为 wFindWhatLen 和 wReplaceWithLen 字段赋值以说明所分配内存的大小。回调函数 lpfnHook()需要同 lpTemplatename 及几个 Flags 标志位合作，以彻底替换掉缺省的 Find/Replace 机制。打个比方，这有点象把您自己的车捎到曼哈顿，再雇个司机开车；我们不会这样做。

FINDREPLACE 结构填好之后，被传递给 FindText()或 ReplaceText()函数。函数可以改变 Flags、lpstrFindWhat、ipstrReplace 几个字段的值；但我说过，无论如何，最好是保留用户所作的设置。

FindText()和 ReplaceText()两个函数分别显示各自的对话框(以非模式对话框的方式)，此时还不会有什么发生。显示的对话框如图 12.3和 12.4所示；但是直到用户单击了“FindNext”、“Replace”或“Replace All”按钮之后，用户自定义消息才会发送到拥有对话框的那个窗口。

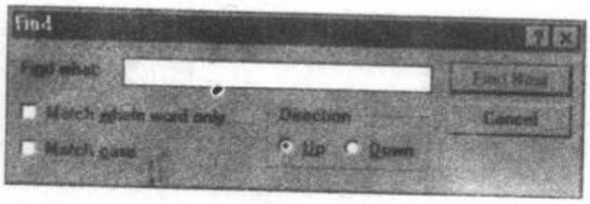

图12.3　通用的 Find 对话框

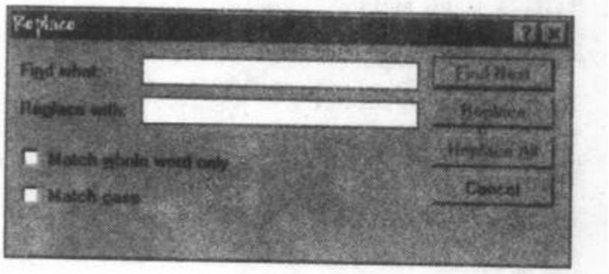

图12.4　通用的 Replace 对话框

如果某个窗口要利用消息做些什么的话，它必须要认得这个消息。为了识别出这个消息，窗口首先应注册它。SDK 的 RegisterWindowMessage()函数可当此任。要注册的字符串已经在 WINDOWS. H 中定义为 FINDMSGSTRING，因而注册工作是很容易的：

FRMsg＝RegisterWindowMessage(FINDMSGSTRING)，

RegisterWindowMessage()接受一个字符串参数，返回的则是一个保证在这个 Windows 会话期间唯一的数值。故它允许多个应用程序同时运行，却不能保证程序在多个 Windows 会话下运行时表示那个字符串的数值都是相同的。也即是说，您不能对这个消息 ID 硬编码，您也不能在 switch 语句中把它作为分支入口，因为 switch 语句要求是常数。然而，改用简单的 if 语句是可行的。

12.3 生成 FindReplace 控件的框架

您现在应该知道怎样利用 VBX Genie 生成 Find 和 Replace 对话框等控件的基本代码，供包括您自己在内的 VB 程序员们所用。有一些选项需要指定；我们来逐页地讲解。

首先，当然是从 C/C++开发环境的工具菜单选择 VBX Genie。第一页显示的是 Project：对选项作一些调整使之如图12.5。

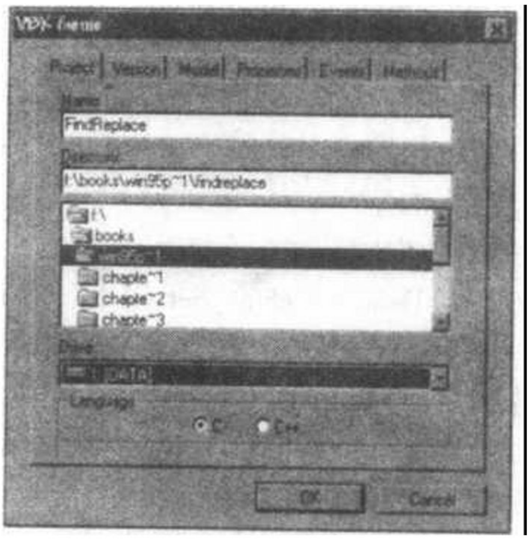

图12.5 VBX Genie 中设定的项目设置选项

当然您也完全可以不必照搬我的设定，而把项目放在另一个盘上！

图12.6所示的页是控件的版本信息。

这个 VBX 是我编写的，所以版权信息就应该是这样。相信您明白这一点，在创建自己的 VBX 时把您的大名放到这页上！

下一个是 Model 页，如图12.7所示。

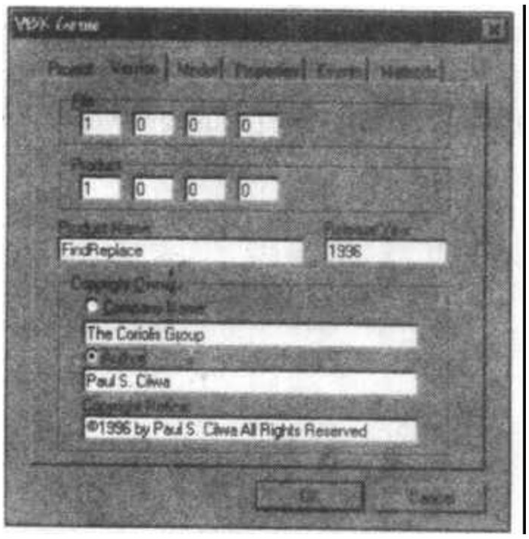

图12.6 设定 FindReplace 控件的版本信息

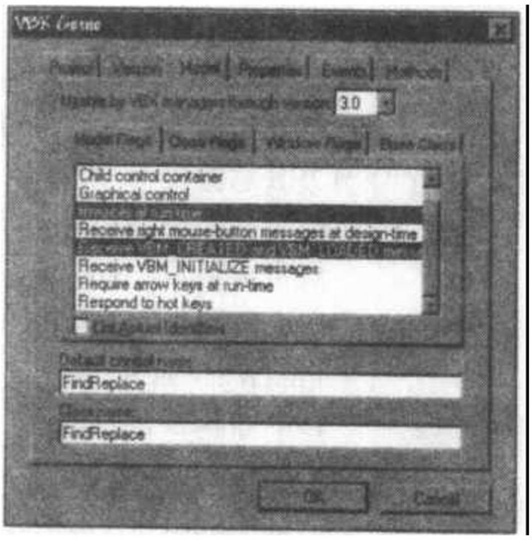

图12.7 Model 页的各项设定

在上面的阐述中没有讲到 Class Flags、Window Flags 和 Base Class 面板，这无关紧要……因为对它们无需作任何改动。

紧接着的页是选定控件所支持的标准属性，在下面列出：

●Name，Index，Parent（必需的）

●HelpContextID

●LeftNoRun，TopNoRun

●Tag

除此之外，FindReplace 控件还需要如下定制属性：

Action　数据类型：Enum
标志位：No runtime read、omit from properties、Set via Message
允许值："Find"、"FindAgain"、"Replace"

Direction　数据类型：Enum
标志位："Default Value"、"Set Direct to Data"
允许值："Up"、"Dowm"
常用值："Up"

EnableMatchCase　数据类型：Boolean
标志位：Default Value、Set via Message
常用值："True"

Enable WholeWord　数据类型：Boolean
标志："Default Value"、"Set via Message"
常用值："True"

EnableDirection　数据类型：Boolean
标志：Default Value、Set via Message
常用值："True"

MatchCase　数据类型：Boolean
标志：Default Value、Set Direct to Data
常用值："False"

WholeWord　数据类型：Boolean
标志：Default Value、Set Direct to Data
常用值："False"

FindReplace 控件没有用到任何标准事件，但它实现了一些用户事件(custom event)；下面列出这些用户事件及其参数：

Find　Direction as Integer
MatchCase as Integer
WholeWord as Integer
FindWhat as String

Replace　Direction as Integer
MatchCase as Integer
WholeWord as Integer
FindWhat as String
ReplaceWith as String

ReplaceAll　Direction as Integer
MatchCase as Integer

```
WholeWord as Integer
FindWhat as String
ReplaceWith as String
```

这个控件不响应任何标准事件，所以在给出了所有这些信息之后，即可单击OK按钮并生成框架！

12.4 存储和访问 FINDREPLACE 结构

对于一些您声明想要在VB里设定的属性，VBX Genie将在VBData结构中为每个属性生成一个数据项。那固然不坏，但我们需要把那些自动生成的项用一个FINDREPLACE结构的项替换掉：

```
typedef struct
  {
FINDREPLACE Data;
} VBDATA, far*LPVBDATA;
```

下面的相关工作是修改那些属性的定义，以便VB把数据放置到正确位置。特别地，有三个属性的值是存储在Flags属性中：

```
PROPINFO Property_Direction=
  {
 "Direction",
DT_ENUM|PF_fDefVal|PF_fGetData|PF_fSaveData|PF_fSetData,
_offsetin(FINDREPLACE,Flags), 0x10,
(long)1,
 "Up\0Down\0", 2
};

PROPINFO Property_MatchCase=
  {
 "MatchCase",
DT_BOOL|PF_fDefVal|PF_fGetData|PF_fSaveData|PF_fSetData,
_offsetin(FINDREPLACE,Flags), 0x12,
(long)FALSE,
NULL, 0
};

PROPINFO Property Whole Word=
  {
 "WholeWord",
DT_BOOL|PF_fDefVal|PF_fGetData|PF_fSaveData|PF_fSetData,
```

```
offsetin(FINDREPLACE,Flags), 0x11,
(long)FALSE,
NULL, 0,
};
```

在这里，我们利用了 Visual Basic 用户控件(custom control)所具有的，而 VBX Genie 所没有直接支持的少数几个特点之一：自动的位存储。这是在 PROPINFO 的第四个字段中指定的，这是一个十六进制数，它的两个半字节位用来说明存储是如何实现的。两个半字节位当中，高位的半字节位告诉 VB 本项存储在多少位中，低位的半字节位指明在第三个参数所指定的偏移范围内，数据位必须左移多少位。

下一项任务是初始化 VbData 结构和注册前面讲到的 Find/Replace 消息。这些都在 OnCreate()函数里完成：

```
#define FINDREPWHATSIZE 256

UINT FindReplaceMessage;
UINT FindReplaceHelpMessage;

static vold near pascal OnCreate
 (
HCTL Control
HWND Window
)
 {
LPVBDATA VbData-(LPVBDATA)VBDerefControl(Control);
VbData->Data. lStructSize=sizeof VbData->Data;
VbData->Data. hwndOwner=Window;
VbData->Data. hInstance=NULL;
VbData->Data. Flags=0;
VbData->Data. lpstrFindWhat=malloc(FINDREPWHATSIZE);
VbData->Data. lpstrFindWhat [0] =0;
VbData->Data. lpstrReplace With=malloc(FINDREPWHATSIZE);
VbData->Data. lpstrReplaceWith [0] =0;
VbData->Data. wFindWhatLen=FINDREPWHATSIZE;
VbData->Data. wReplaceWithLen=FINDREPWHATSIZE;
VbData->Data. lCustData=0;
VbData->Data. lpfnHook=NULL;
VbData->Data. lpTemplateName=NULL;

FindReplaceMessage=RegisterWindowMessage(FINDMSGSTRING);
FindReplaceHelpMessage=RegisterWindowMessage(HELPMSGSTRING);
```

```
}
```

lpstrFindWhat 和 lpstrReplaceWith 字段的缓冲区由 malloc()分配；它们得在 OnDestroy()中释放掉：

```
static void near pascal OnDestroy
    (
    HCTL Control
    )
 {
LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
free(VbData->Data.lpstrFindWhat);
free(VbData->Data.lpstrReplaceWith);
}
```

我们注过册的那些消息必须由控件过程捕获(catch)和发送(dispatch)；最好是在过程的开头做这件事：

```
long far pascal_export CtlProc(HCTL Control,HWND Window,USHORT Msg,USHORT
wParam,long lparam)
 {
long Error=0;

if(Msg==FindReplaceMessage)
    {
    OnFindReplace(Control,(FINDREPLACE far*)lParam);
    return 0;
    }
```

OnFindReplace()是第一个消息的处理程序，用户事件就是在这里触发的。

要触发事件，先得准备好事件处理程序的参数。有的参数是以位的方式存储在 FINDREPLACE 的 Flags 字段中，而 Visual Basic 不能识别这些位。其他的参数则是以 C 风格的字符串的方式存储，而 Visual Basic 同样也不能识别字符串。所以在触发事件之前，这些参数必须要转换成 VB 的数据类型。在处理完事件之后，VB 字符串参数的句柄必须要释放掉。

这些工作由下面的代码完成：

```
static void near pascal OnFindReplace(HCTL Control,FINDREPLACE far*Data)
 {
int Direction;
int MatchCase;
int WholeWord;
struct
    {
    HLSTR Replace With
```

```
    struct
        {
        HLSTR FindWhat;
        LPINT WholeWord;
        LPINT MatchCase;
        LPINT Direction;
        LPVOID Index;
        } Find;
    } Replace;

Replace. ReplaceWith=VBCreateHlstr(Data->lpstrReplaceWith,lstrlen(Data->lpstrReplaceWith));
Replace. Find. FindWhat=VBCreateHlstr(Data->lpstrFindWhat,lstrlen(Data->lpstrFindWhat));
Replace. Find. WholeWord=&WholeWord;
Replace. Find. MatchCase=&MatchCase;
Replace. Find. Direction=&Direction;

WholeWold-Boolean [((Data->Flags & FR_WHOLEWORD)!=0)];
MatchCase=Boolean[((Data->Flags & FR_MATCHCASE)!=0)];
Direction=((Data->Flags &FR_DOWN) ! =0);

if(Data->Flags & FR_FINDNEXT)
    VBFireEvent(Control. IEVENTINFO_Find,&(Replace. Find));
else if(Data->Flags & FR_REPLACE)
    VBFireEvent(Control. IEVENTINFO_Replace,&Replace);
else if(Data->Flags & FR_REPLACEALL)
    VBFireEvent(Control. IEVENTINFO_ReplaceAll,&Replace);

VBDestroyHlstr(Replace. ReplaceWith);
VBDestroyHlstr(Replace. Find. FindWhat);
}
```

注意其中 Boolean 数组的用法，不需多大开销，就完成了从 C 语言的布尔型值到 Visual Basic 的布尔型值之间的转换。

多数属性是通过 VBM_SETPROPERTY 消息设置的，在函数 OnSetProperty()中完成对该消息的处理。函数框架由 VBX Genie 生成；而我们得在框架中填入代码。Action 属性由一个辅助函数处理；其他属性是位的形式，必须依据调用的入口值或设置、或清除：

```
static BOOL,near pascal OnSetProperty (HCTL Control,HWND Window,USHORT PropertylX,
```

```
        long Value, long far *Error)
    {
    LPVBDATA VbData= (LPVBDATA) VBDerefControl(Control);
    switch(PropertyIX)
        {
        case IPROPINFO_Action:
            OnAction(Control,(int)Value);
            break;
        case IPROPINFO_EnableMatchCase:
            if(Value)
                VbData->Data.Flags &=(~FR_NOMATCHCASE);
            else
                VbData->Data.Flags|=FR_NOMATCHCASE;
            break;
        case IPROPINFO_EnableDirection:
            if(Value)
                VbData->Data.Flags &=(~FR_NOUPDOWN);
            else
                VbData->Data.Flags|=FR_NOUPDOWN;
            break;
        case IPROPINFO_EnableWholeWord:
            if(Value)
                VbData->Data.Flags &=(~FR_NOWHOLEWORD);
            else
                VbData->Data.Flags|=FR_NOWHOLEWORD;
            break;
        default:
            return FALSE;
        }
    return TRUE;
    }
```

与在Common Dialog、OLE2和其他控件中一样，Action属性是一个枚举的属性，它促使特定的动作发生。如果VBX的原设计者们不曾不可思议地对创建用户方法(custom method)的能力加以阻拦，本可以不要“Action”属性的，因为我们可以自己创建用户属性和事件。但既然事已至此，“Action”就得发挥作用了。

设置这个属性(因而也引发了一些事情，如显示出Find或Replace对话框)的处理程序是OnAction()。这一属性有三个可能值，分别是Action_Find。Action_FindAgain和Action_Replace：

```
    static void near pascal OnAction (HCTL Control,int Action)
    {
```

```
LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
switch(Action)
    {
    case Action_Find:
        FindText(&(VbData->Data));
        break;
    case Action FindAgain;
        if(!VbData->Data.lpstrFindWhat[0])
            FindText(&(VbData->Data));
        else
            {
            VbData_>Data. Flags &=(~FR_DIALOGTERM);
            VbData_>Data. Flags|=FR_FINDNEXT;
            OnFindReplace(Control,&(VbData->Data));
            }
        break;
    case Action_Replace;
        ReplaceText(&(VbData->Data));
        break;
    };
}
```

Action_Find 只需简单调用通用对话框 DLL 的 FindText()函数。类似地，Action_Replace 调用 ReplaceText()。二者中最有意思的要算 Action FindAgain，如果没有指定查找文本，那么它与 Action_Find 的表现是一样的。而如果在 FINDREPLACE 结构中有查找文本，我们希望同样的事情发生，就如同 Find 对话框是可见的，而用户刚好又单击了“Find Next”按钮一样。

因此在代码中，我们检查 lpstrFindWhat 的第一个字节。如果它是 NULL，我们知道还没有指定查找文本，于是调用 FindText()。如果不是，即是指定了文本；我们就直接调用例程 OnFindReplace()，而该例程或许已被 Find 和 Replace 对话框发出的消息所调用。

OnGetProperty()为响应 VBM_GETPROPERTY 消息而调用，它与 OnSetProperty()类似，当然有一点不同，Action 属性是只写属性，因而在 OnGetProperty()中没有考虑处理 Action 属性：

```
static BOOLnear pascal OnGetProperty (HCTL Control,HWND Window,USHORT PropertyIX,
                LPVOID Value,long far' Error)
  {
  LPVBDATA VbData= (LPVBDATA) VBDerefControl(Control);
  switch(PropertyIX)
    {
  case IPROPINFO_EnableMatchCase;
```

```
        (*(long*)Value)=(!(VbData->Data.Flags & FR_NOMATCHCASE));
        break;
    case IPROPINFO_EnableDirection:
        (*(long*)Value)=(!(VbData->Data.Flags & FR_NOUPDOWN));
      break;
    case IPROPINFO_EnableWholeWord:
        (*(long*)Value)=(!(VbData->Data.Flags & FR_NOWHOLEWORD));
        break;
    default:
        return FALSE;
            }
        return TRUE;
    }
```

从磁盘装入属性的工作被简化了，因为我们只接收三个属性的消息，而这三个属性都是布尔型的值。这样，我们可以用单个的字节保存它们。我们也可以用OnSetProperty()函数对这些装入值实际地加以处理，以达到进一步节省代码的目的：

```
static BOOL near pascal OnSaveProperty(HCTL Control,HWND Window,USHORT PropertyIX,
    HFORMFILE FormFile,long far * Error)
  {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  long Value;
  char Byte;
  switch(PropertyIX)
      {
  case IPROPINFO_EnableMatchCase:
  case IPROPINFO_EnableDirection:
  case IPROPINFO_EnableWholeWord:
      OnGetProperty(Control,Window,PropertyIX,&Value,Error);
      Byte=(char)Value;
      *Error=VBWriteFormFile(FormFile,&Byte,sizeofByte);
      break;
  default
        return FALSE;
      }
  return TRUE;
  }
```

同样，为了要从磁盘装入一个属性，除了真正地读入表文件之外，其他的工作都可以调

用 OnSetProperty()函数完成：

```
    static BOOL near pascal OnLoadProperty(HCTL Control,HWND Window,USHORT
PropertyIX,
        HFORMFILE FormFile,long far * Error)
    {
    LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
    char Value;
    switch(PropertyIX)
        {
        case IPROPINFO_EnableMatchCase:
        case IPROPINFO_EnableDirection:
        case IPROPINFO_EnableWholeWord:
            *Error=VBReadFormFile(FormFile,&Value,sizeof Value);
            OnSetProperty(Control,Window,PropertyIX,Value,Error);
            break;
        default:
            return FALSE;
        }
    return TRUE;
    }
```

请注意我们是怎样以单个字节的方式保存布尔型变量的，尽管在代码中它多是作为长整型使用的。尽管在本控件中没有用到，但仍请您思考一下另外一种节省存储空间的技巧：在单个字节内存储多达8个属性。在响应某个属性的消息时，实际地读/写所有8个属性。对于其他的7个属性的存储或装入消息，只需简单地返回 TRUE。Visual Basic 就是采用这种方法把标准的Top、Left、Bottom 和 Right 属性存储在一个单元里。

12.5 对上下文敏感的联机帮助提供支持

还剩下一块拦路石没有搬开：实现用户联机帮助。如果程序员给出一个 HelpContextID 的属性值(非0)，那么 Find 和 Replace 对话框将显示 Help(帮助)按钮；按动这个按钮将得到与应用程序相关联的联机帮助，所选主题的序号与 HelpContextID 值刚好匹配。如果没有给出HelpContextID 的话，就根本不会显现 Help 按钮。

通过指定 MODEL_fLoadMsg 标志位，我们可以做到这一点。有了这个标志位。就可以在设定了控件的全部属性之后，发出一个 VBM_LOADED 消息；这样我们就可以利用此消息查看 HelpContextID 属性的值为多少。这个消息当然是在控件过程的 switch 语句中捕获并处理的：

```
    switch(Msg)
        {
            /
```

```
        /
    case VBM_LOADED;
        OnLoaded(Control);
        break;
```

如上所述，函数 OnLoaded()检查 HelpContextID 属性的值，以此决定是否要把相应的标志位(即 FR_SHOWHELP)同 FINDREPLACE 结构的 Flags 字段进行 OR 运算。

```
static void near pascal OnLoaded(HCTL Control)
  {
long Value;
VBGetControlProperty(Control,IPROPINFO_STD_HELPCONTEXTID,&Value);
if(Value != 0)
    {
    LPVBDATA VbData= (LPVBDATA) VBDerefControl(Control);
    VbData->Data Flags|=FR_SHOWHELP;
    }
}
```

现在，当对话框把帮助的消息发送到控件过程时，它只需把消息转换成 F1键的 WM_KEYDOWN，再由 Visual Basic 响应：

```
long far pascal _export CtiProc
    (
    HCTL Control,
    HWND Window,
    USHORT Msg,
    USHORT wParam,
    long lParam
    )
  {
long Error=0;

if(Msg==FindReplaceMessage)
    {
    OnFindReplace(Control,(FINDREPLACE far*)lParam);
    return 0;
    }

if(Msg==FindReplaceHelpMessage)
    {
    SendMessage(Window,WM_KEYDOWN,VK_F1,0);
    return 0;
    }
```

```
switch(Msg)
    /
    /
```

12.6 控件的发行

我经常说，直到联机帮助也做好了，控件才算是大功告成。由于制作联机帮助不是本书的重点，我在这里不想告诉您怎样创建这样的文档。但有一条：千万要包括一页常量，以便程序员能拷贝到剪贴板。对于 FindReplace 控件来说，总共只有5个常量：

```
Action property
Const FR_FindFirst=1
Const FR_FindNext=2
Const FR_Replace=3
Direction property
Const FR_Up=0
Const FR_Down
```

12.7 测试 FindReplace 控件

为了测试 FindReplace 控件，需要创建一个小型的工程，其中仅有 FindReplace 这个唯一的控件。把我们的新控件放置到表上，并加入菜单，如图12.8所示。

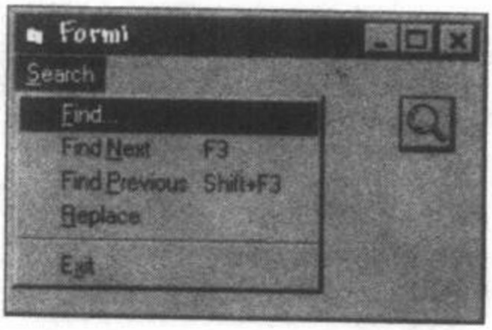

图12.8 加入 Search 菜单

处理菜单命令的代码实现如下：

```
Sub mnu_SearchExit_Click()
Unload Me
End Sub

Sub mnu_SearchFind_Click()
FindReplacel. Action=FR_FindFirst
End Sub

Sub mnu_SearchFindNext_Click()
```

```
FindReplace1. Direction=FR_Down
FindReplace1. Action=FR_FindNext
End Sub

Sub mnu_SearchFindPrevious_Click()
FindReplace1. Direction=FR_Up
FindReplace1. Action=FR_FindNext
End Sub

Sub mnu_SearchReplace_Click()
FindReplace1 Action=FR_Replace
End Sub
```

命令处理程序的各条语句颇为典型，它们应该(而且正是)包括进控件的帮助文件中，这样，在实现各项功能时您只需把它们复制和粘贴到应用程序的代码中。

还可以为控件事件提供事件处理程序，比如：

```
Sub FindReplace1_Find(Direction As Integer,MatchCase As Integer,WholeWord As Integer,FindWhat As String)
MsgBox "Asked to find"+FindWhat+""
End Sub

Sub FindReplace1 Replace All(Direction As Integer,MatchCase As Integer,WholeWord As Integer,FindWhat As String,ReplaceWith As String)
MsgBox "Asked to replace"+FindWhat+"with"+ReplaceWith+""
End Sub
```

现在请试运行这个应用程序。试一下各个命令。在您给出或者不给出 HelpContextID 时，观察一下 Find 和 Replace 对话框的表现有何不同。(当然如果给出的 HelpContextID 对应用程序的帮助文件来说是无效的话，那么 VB 就不能确实地显示一个帮助主题。)

把 FindReplace VBX 添加到工具箱中，您会发现，要编写出满足 Windows 用户所期待的所有功能的 VB 应用程序，就变得容易多了。

第十三章　为标准控件加上工具提示

我们人类喜欢给事物贴上标签。如果没有标签，我们就感到不舒服，或许这正好解释了"工具提示"（即对 Windows 应用程序中按钮和其他控件作出标识的那些小标签）存在的必然性。在 Microsoft 公司威吓我们、要把我们赶进他们营造的32位世界的许多恼人的手段中，其中有一个就是说他们已经拒绝为16位 Visual Basic 应用程序提供工具提示的支持。好吧，在这一章里，我们就来自力更生。

13.1　时代的标志

您知道，所谓工具提示，是指那些小标签，当鼠标光标在某个位图按钮上停留一段时间时，它们就跳出来，目的是使按钮更醒目、更明白。

实现工具提示最直接的办法当然是在按钮中就设计好这样一个能力。也即是说，抛弃 Visual Basic 自带的按钮不用，而创建您自己的、有内嵌的工具提示功能的定制按钮。这当然也可以在 VBX 中完成。但是它不会为其他任何控件类型提供工具提示功能的支持，而有时这恰恰是一个不错的功能。

另一个办法是编写 VB 代码。这相当容易；因为每个控件都有 GetFocus 和 KillFocus 事件。在两者之间，您可以加入一个 Timer（计时器）控件和一个浮动 Label（标签）控件。这样的话，您就可以实现在任何地方显示一个工具提示。但是，如果这以一种更为面向对象的方式来实现，不是会更好吗？

我的 VBX 解决方案适用于任何定制控件。甚至是 OCX 控件，只要它有 hWnd 这样一个属性（它只省略了象 Image 和 Label 这样的图形控件）。它将以容器对象的面目出现，您可以往其中加入标题（用 AddItem 方法）。而且，由于它是 VBX，您可以在 Visual Basic 3.0版本或者是4.0的16位版本中使用它。

13.2　路标：一直前行

Tips 控件以其某些古怪的要求而称得上是一个很有意思的工程。我们将得创建一个新的窗口类，把已有的窗口类（包括 OLE 服务程序的窗口！）作为子类，甚至"欺骗"Visual Basic 把窗口句柄当成数组下标来用。

从程序员/用户的角度看，如果某个表包含了按钮及其他控件，而这些控件需要有工具提示，就可以用到 Tips 控件。如果终端用户的鼠标光标越过控件，提示窗口就显示出来。

Tips 控件实现的思路是，您可以使用 AddItem 方法往其中加入一些控件/标题。控件由其 hWnd 属性确定，而这个属性正好是一个数字型的值；因此我们可以把各个属性作为下标来使用。也即是说，把一条提示添加到 Tips 控件的句法看起来象这样：

```
Tips AddItem "Print the current document", btn_Print.hWnd
```

这样，一个 Tips 控件就必须维护加入到其中的各个控件所形成的列表。

当把一个控件加入到这个控件集合时，Tips 控件会把它作为子类。为此，所有发送给原有

控件窗口过程的消息将首先被 Tips 控件的窗口过程截获。提示窗口所提供的额外动作有两方面：其一，当鼠标光标移动到它上面时，控件会察觉到，并启动一个计时器。(如果光标移开，计时器就被终止。)其二，如果计时器超时而鼠标光标仍定在控件上面，就创建并显示一个弹出式的提示窗口。

为此，必须编写这个提示窗口类。我们需要的窗口类应能根据其正文内容作大小调整，这一点是 Static 类所不具备的。那就行了；可调整大小的文本类是本工程的次要部分，而且它是可重用的。

13.2.1 选用什么语言作标记

我们将利用 VBX Genie 来启动这一工程。工程起名为“Tips”。下一步要难些：我们需要决定用哪种语言编写这个 VBX。我建议用 C 语言，因为我们得把其他控件窗口作为子类，而 C 语言提供给我们更好的控制能力，这正是我们所需的。

“Invisible at runtime”是我们唯一需要的 Model 标志位。标准属性我们用到的有 BackColor、ForeColor、Enabled、字体的有关属性、TopNoRun、LeftNoRun 以及必需的 Name、Index 和 Parent 属性。

用户属性有：

●Delay：short 变量，表示毫秒数，鼠标必须在控件上停留 Delay 毫秒之后，工具提示才会显现。

●Captions：该属性是字符串数组，它允许在运行时改变标题。

●AutoColor：布尔型属性，用以指明工具提示窗口的颜色是从预定义的系统颜色中获得，还是从标准属性 ForeColor 和 BackColor 中获得。

Tips 控件不支持事件(即使是标准事件)，而您会要求对 AddItem、RemoveItem 和 Clear 方法的支持。

工程生成以后，我们还得修改其中的一个文件：VISUAL.C。这个文件的开头是：

```
#include "internalh"
#include <string.h>

#define segment(p)\
    ((HANDLE)(((unsigned long)(void far*)(p)) >>16L))
#define_offsetin(struc,fld)((USHORT)&(((struc*)0) ->fld))
#define VBERR_BADINDEX 381

static long Boolean [2] = {0, -1};
```

VISUAL.C 的下一个代码块是 VbData 结构：

```
#define MAXTOOLS 64
typedef struct
  {
  HWND TipParent;
  HWND TipWindow;
```

```
  HFONT TipFont;
  UINT Delay;
  BOOL AutoColor;
  HBRUSH AutoBrush;
  BOOL Enabled;
  BOOL MouseCaptured;
  int ToolsCount;
  struct
     {
    HSZ Caption;
    HWND Window;
    WNDPROC WndProc;
    } Tools [MAXTOOLS];
  } VBDATA, far*LPVBDATA;
```

按通常的做法，VbDala结构为控件的公共属性和内部属性都留出了空间。需要特别说明的是#deinfe MAXTOOLS。我个人认为在一个表上支持64个按钮已经足够。如果您不同意我的观点，您尽可以把它改成一个大些的数值。

生成的各属性结构列出如下：

```
PROPINFO Property_About=
   {
   "(About)",
  DT_HSZ|PF_fGetMsg|PF_fNoRuntimeW|PF_fGetHszMsg,
  0, 0, 0, NULL, 0
  };

PROPINFO Property_Delay=
   {
   "Delay",
  DT_SHORT|PF_fSetData|PF_fGetData|PF_fSaveData,
  _offsetin(VBDATA,Delay), 0,
  0L,
  NULL, 0
  };

PROPINFO Property_Captions=
   {
   "Captions",
  DT_HSZ|PF_fPropArray|PF_fSetMsg|PF_fGetMsg,
  0, 0,
```

```
    0L,
    NULL, 0
    };

PROPINFO Property_AutoColor=
    {
    "AutoColor",
    DT_BOOL|PF_fSetData|PF_fGetData|PF_fSaveData,
    _offsetin(VBDATA,AutoColor), 0,
    0L,
    NULL, 0
    };

PPROPINFO Properties [] =
    {
    PPROPINFO_STD_NAME,
    PPROPINFO_STD_INDEX,
    PPROPINFO_STD_BACKCOLOR,
    PPROPINFO_STD_ENABLED,
    PPROPINFO_STD_FONTBOLD,
    PPROPINFO_STD_FONTITALIC,
    PPROPINFO_STD_FONTNAME,
    PPROPINFO_STD_FONTSIZE,
    PPROPINFO_STD_FONTSTRIKE,
    PPROPINFO_STD_FONTUNDER,
    PPROPINFO_STD_FORECOLOR,
    PPROPINFO_STD_TOPNORUN,
    PPROPINFO_STD_LEFTNORUN,
    PPROPINFO_STD_PARENT,
    PPROPINFO_STD_VISIBLE,
    PPROPINFO_STD_LAST,
    &Property_About,
    &Property_Delay,
    &Property_Captions,
    NULL
    };

enum
    {
    Index_Property_Name,
```

```
Index_Property_Index,
Index_Property_BackColor,
Index_Property_Enabled,
Index_Property_FontBold,
Index_Property_FontItalic,
Index_Property_FontName,
Index_Property_FontSize,
Index_Property_FontStrike,
Index_Property_FontUnder,
Index_Property_ForeColor,
Index_Property_TopNoRun,
Index_Property_LeftNoRun,
Index_Property_Parent,
Index_Property_Visible,
Index_Property_Last,
Index_Property_About,
Index_Property_Delay,
Index_Property_Captions,
Index_Property_End
};
```

Tips 控件并不触发任何事件，故 Events 数组只需为：

```
PEVENTINFO Events [] =
  {
  NULL
  };

enum
  {
  Index_Event_End
  };
```

13.2.2 为方法编写代码

尽管我们把各控件的窗口句柄当成是 Tips 控件 Captions 数组的下标来使用，但它们并不是真正的下标。我们可以让控件的使用者把窗口句柄当成是下标，而如果是这样的话，我们就得把句柄转换成真正的数组下标。FindIndex () 函数就是用来完成这一使命的：

```
static int near pascal FindIndex (HWND Window,LPVBDATA VbData)
  {
  register i;
```

```
for(i=0;i<VbData->ToolsCount;i++)
    if(VbData->Tools[i],Window==Window)
        returni;
return-1;
}
```

VBX Genie 常常在方法处理函数前放上一个结构定义，函数(onMethod())在后面要用到这个结构，以解释传递给该函数的参数含义：

```
typedefstruct
  {
  long Count;
  HSZ Item;
  long Index;
  } ADDITEM;
typedef ADDITEM far* LPADDITEM;
```

在 OnMethodAddItem()的实际代码之前，我们需要给出工具窗口子过程的说明，稍后我会解释其原因：

```
LRESULT_export FAR PASCAL ToolWndProc
    (
    HWND Window,
    UINT Msg,
    WPARAMwParam,
    LPARAMlParam
    );
```

现在我们可以看看方法的代码了：

```
LRESULT near pascal OnMethodAddItem(HCTL Control,HWND Window, LPSTR
                                              Buffer,HWND Tool)
  {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  register i=FindIndex(Tool,VbData);

  if((i>-1)||(VbData->ToolsCount==MAXTOOLS))
      return VBERR_BADINDEX;

  i=(VbData->ToolsCount)++;
  VbData->Tools [i]. Caption=
      VBCreatellsz(segment(Control), Buffer);
  VbData->Tools [i]. Window=Tool;
```

```
VbData->Tools [i]. WndProc=
  (WNDPROC)GetWindowLong(Tool,GWL_WNDPROC);
SetWindowLong(Tool,GWL_WNDPROC,(long)ToolWndProc);
SetProp(Tool,"TipControlH",
  (HANDLE)HIWORD((long)Control));
SetProp(Tool,"TipControlL",
  (HANDLE)LOWORD((long)Control));
SetProp(Tool,"TipIndex",(HANDLE)i);
return 0;
}
```

在间接引用控件句柄以取得指向 VbData 的指针后，函数检查本窗口是否已经被加入列表。如果已加入，或列表空间已经不够，函数就返回一个错误。否则，我们就把该工具的提示窗口要显示的标题内容存入其中；这需要调用 VBCreateHsz()以创建一个新的 HSZ 句柄。然后把工具窗口自身的句柄加入其中，以便以后 FindIndex()能找到它。

最后，我们着手把加入列表的工具变为子类。

13.2.3 子类方法的原理

子类方法是一种 SDK 技术，您可以在消息传递机制和某个窗口的原有窗口过程之间插入另一个窗口过程。这样，您就拥有了对所经过的消息首先进行处理的权力，可以添加、禁止或修改其行为表现。

遇到子类窗口过程代码时，我再对它作解释；现在只需接受有这么一种方法，它可以完成上述使命。剩下的就是如何产生子类的实际工作了。

首先要获得窗口的原始窗口过程的地址。以 GWL_WNDPROC 为索引调用 GetWindowLong()函数可做到这一点。得把这个地址保存起来，因为子类窗口过程还会用到它，以保持那些不需改变的消息处理链的完整性。

有了这个地址，就可以调用 SetWindowLong()把它替换成您自己的子类过程地址。不要用 GetClassLong()：它会把所有属于此类的窗口变为子类，而不仅仅是您所感兴趣的那一个。而现在，您面临着如何与子类窗口通信的问题。或者说，是它与您通信的问题。您需要设置窗口的其他一些属性，而且肯定还不能使用窗口的额外字节(您不知道那究竟有多少字节)，况且它们很可能以某种方式在被使用。好在 Windows 提供了另一种权宜之计，可以把属性赋予某个窗口：SetProp()。这个函数接受一个窗口句柄，一个大小写敏感字符串和一个 HANDLE 数据类型的16位数值为参数。这个数值被赋予窗口，与字符串相联系，并且可以通过调用 GetProp()查询。

有一点小麻烦，HCTL 是32位的，而这些“属性”被限定为16位。这就是我在上面的代码当中把它分开并保存为两个属性的原因。

当用户调用方法 RemoveItem，我们必须取消子类窗口，把它恢复回去：

```
typedef struct
  {
  long Count;
```

```
    long Index;
    } REMOVEITEM;
typedef REMOVEITEM far * LPREMOVEITEM;

LRESULT near pascal OnMethodRemoveItem(HCTL Control,HWND Window,HWND
Tool)
    {
    LPVBDATA VbData=(LPVBDATA) VBDerefControl (Control);
    register i=FindIndex (Tool,VbData);

    if(i>-1)
        {
        if (IsWindow (VbData->TipWindow))
            DestroyWindow (VbData->Tipwindow);
        VBDestroyHsz (VbData->Tools[i]. Caption);
        SetWindowLong (VbData->Tools[i]. window,GWL_WNDPROC,
            (long) VbData->Tools [i] WndProc);
        RemoveProp (Tool,"TipControlH");
        RemoveProp (Tool,"TipControlL");
        RemoveProp (Tool,"TipIndex");
        if (i<--(VbData->ToolsCount))
            {
            VbData->Tools [i] =VbData->Tools [VbData->ToolsCount];
            SetProp (VbData->Tools[i]. Window,"Index",(HANDLE)i);
            }
        return 0;
        }
    else
        return VBERR_BADINDEX;
    }
```

当一个提示窗口正在显示时，如果上面的代码被执行，窗口就被清除(destroy)。然后恢复原有的窗口过程地址(通过调用SetWindowLong())，并取消先前赋予它的属性。接下来的只是些简单的善后工作，把该元素从Tips数组中删除，用到的办法是把最后一个数组元素移到此位置，并让TipsCount减1。此时，最后这个元素的“Index”属性需作修改，以反映出变化了的新值。SetProps()可随时在需要时调用，以改变任何现有的值，或加入新值。

方法Clear的实现很类似：

```
LRESULT near pascal OnMethodClear (HCTL Control,HWND Window)
    {
    LPVBDATA VbData=(LPVBDATA) VBDerefControl (Control);
```

```
register i;
for (i=0;i<VbData->ToolsCount,i++)
    {
    if(IsWindow (VbData->TipWindow))
        DestroyWindow (VbData->TipWindow);
    if (IsWindow( VbData->Tools[i]. Window))
        {
        SetWindowLong (VbData->Tools[i]. Window,GWL_WNDPROC,
            (long) VbData->Tools [i].  WndProc);
        RemoveProp (VbData->Tools[i],Window"TipControlH");
        RemoveProp (VbData->Tools[i],Window"TipControlL");
        RemoveProp (VbData->Tools[i],Window"TipIndex");
        }
    VBDestroyHsz (VbData->Tools[i]. Caption);
    }
VbData->ToolsCount=0;
return0;
}
```

函数OnMethod()由 VBX Genie 生成，它把不同的方法调用请求传送给相应的处理程序：

```
static long near pascal OnMethod(HCTL Control HWND Window,USHORT Method,
                    void far * Args)
  {
  LRESULT Result=0;
  switch (Method)
      {
      case METH_ADDITEM:
          {
          LPADDITEM AddItem=Args;
          LPSTR Item=VBLockHsz(AddItem->Item);
          int Index=(AddItem->Count>2)?
              (int)AddItem->Index: -1;
          Result=OnMethodAddItem(Control,
              Window,Item,(HWND)Index);
          VBUnlockHsz(AddItem->Item);
          }
          break;
      case METH_CLEAR:
          Result=OnMethodClear (Control,Window);
          break;
      case METH_REMOVEITEM:
```

```
            {
            LPREMOVEITEM RemoveItem=Args;
            Result=OnMethodRemoveItem(Control,Window,
                (HWND)RemoveItem->Index);
            }
            break;
        default:
            Result=VBDefControlProc(Control,
                Window,VBM_METHOD,Method,(long)Args);
        }
    return Result;
    }
```

13.2.4 设置属性

Tips 控件所支持的多数属性都由 VBX 管理器直接写入 VbData 结构的标准属性或用户属性。然而有一个不是：这就是 Captions 属性，它是一个数组，允许用户在标题和窗口已经通过使用 AddItem 方法设置好之后，再用编程的手段改变提示窗口要显示的正文内容。属性可以通过程序设定：

```
static BOOLnear pascal OnSetProperty(HCTL Control,HWND Window,USHORT PropertyIX,
            long Value,long far*Error)
    {
    LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
    LPDATASTRUCT Data=(LPDATASTRUCT)Value;
    switch(PropertyIX)
        {
        case Index_Property_Captions:
            {
            register i=
                FindIndex((HWND)Data->index [0]. data, VbData);
            if(i>-1)
                {
                VBDestroyHsz(VbData->Tools[i].Caption);
                VbData->Tools [i]. Caption=
                    VBCreateHsz(segment(Control),
                        (LPSTR)Data->data);
                if(VbData->TipWindow)
                    InvalidateRect(VbData->TipWindow,NULL,TRUE);
                }
            else
```

```
            *Error=VBERR_BADINDEX;
        }
        break;
    default:
        return FALSE;
    }
return TRUE;
}
```

由于 Captions 是数组属性，那么 Value 自己就是指向 DATASTRUCT 的指针，而不是所要的值。在一个新的 HSZ 字符串被创建之前，先前在这个位置上的那个由 VBX 管理的 HSZ 字符串标题必须要被清除掉(实际上是释放)。

此外，还可以查询一个标题：

```
static BOOL near pascal OnGetProperty(HCTL Control,HWND Window,USHORT Prop-
                                      ertyIX,
                                      LPVOID Value,long far* Error)
 {
 LPVBDATA VbData=(LPVBDATA) VBDerefControl (Control);
 LPDATASTRUCT Data=(LPDATASTRUCT)Value;
 switch (PropertyIX)
     {
     case Index Property Captions:
         {
         register i=
             FindIndex ((HWND) Data->index [0]. data,VbData);
         if(i >-1)
             *((HSZ far*)Data→data) =
                 VBCreateHsz (segment (Control),
                     VBDerefHsz(VbData->Tools[i]. Caption));
         else
             *Error=VBERR_BADINDEX;
         }
         break;
     default:
         return FALSE;
     }
 return TRUE;
 }
```

尽管我们把标题保存为 HSZ 格式，并且那也正是我们所想要的，但是我们不能只是复制 HSZ 句柄。相反，我们得创建一个新的字符串；当不再需要它时，VBX 管理器负责清除它。

我们并没有对 OnGetStringProperty()函数作修改，所以这里不再讲解它。下面我们要看看几个由控件过程调用的事件处理程序，以及它们的辅助函数。首先要看的是有关提示窗口显示颜色的函数。

13.2.5 VB 中的系统颜色

我们知道，标准属性 ForeColor 和 BackColor 并没有提供多少颜色映射表以辅助用户设置特定的前景色和背景色。我们还知道，属性栏中的颜色是以十六进制长整型值表示的。而且您也许还知道，颜色是用三个连续字节表示的，分别代表红色值、绿色值和蓝色值，此外还有一个字节未用。

问题来了，假如是高位字节未用，那为什么颜色值不总是0x00，除非您显式地改变它?例如，某个标签控件的 ForeColor 属性的初始值为：

&H80000012&

答案是，VB 中的颜色，其含义比 Windows SDK 中简单的 COLORREF 有所增强。未用的高位字节也被派上了用场：如果其值为0x80，则表示：本颜色属于系统颜色，把低位字节作为索引值调用函数 GetSysColor()，即可得到所需的颜色。上面的示例即是 COLOR_WINDOWTEXT 索引值，它表示的是控件或表中普通正文的颜色。用调色板是不能返回或指定一个系统颜色的，但您完全可以直接键入这个颜色值。

Windows 95 比它的先辈们提供了更丰富的系统颜色。比如，供工具提示框和工具提示正文使用的两类颜色。它们分别是23号系统颜色和24号系统颜色(十进制)。GetSysColor()同样适用于它们。因此您会想，我们为何不把 ForeColor 属性设置为 &H80000018&，这样就自动地得到了我们想要的颜色?错!16位版本的 VB4拒绝接受这个数值，而把它改为 &H80000005&(他们并没有充分地陈述理由)。并且，尽管 VB 接受 &H80000017& 这个值，但是它返回的却是纯黑。因此我们被迫把颜色设置的工作交给控件来完成，而属性 AutoColor 则负责把这个特性打开。或者，如果 VB 程序员希望自己为用户指定颜色的话，就把这个特性关闭。

这相当于把球踢给了 Tips 控件。确实，如果在 Windows 95 下我们只需要简单地以23和24为索引调用 GetSysColor()，就能够得到工具提示窗口的用户自定义颜色值。然而，在 Windows3.1中这些颜色并未定义，要是应用程序是运行在 Windows 3.1中，又会怎样呢?对未定义的颜色，GetSystemColor()返回黑色。我们可以利用这一点来暗示索引值已经是在 COLOR_WINDOWTEXT 和 COLOR_WINDOW 之后了。但这只能是在两个值都是黑色的条件下，因为用户很有可能选择正文或工具提示窗口为黑色，但不能都是黑色。

这些想法封装在函数 GetInfoTextColor()和 GetInfoBkColor()中：

```
static COLORREF near pascal GetInfoTextColor (void)
    {
    COLORREF Result=GetSysColor (23);
    if ((Result==0)&&(GetSysColor(24)==0))
        Result=GetSysColor(COLOR_WINDOWTEXT);
    return Result;
    }

static COLORREF near pascal GetInfoBkColor (void)
```

```
{
COLORREF Result=GetSysColor(24);
if((Result==0)&&(GetSysColor(23)==0))
    Result=GetSysColor(COLOR_WINDOW);
return Result;
}
```

在任何用到 ForeColor 和 BackColor 这两个标准属性的控件中，您都需要发送 WM_CTLCOLOR 消息给父窗口。该消息改换设备环境(通过 wParam 传递)的文本颜色，并返回被选中背景色的刷子。

如果用户选定了 AutoColor 属性，就应该出 Tips 控件提供那个刷子。现在我们不能对刷子放任不管了，因为它是归我们所有，最后还要由我们负责删除。这样，在创建 Tips 控件的同时还要创建刷子：

```
static void OnCreate (HCTL Control,HWND Window)
  {
  LPVBDATA VbData= (LPVBDATA) VBDerefControl (Control);
  VbData->TipParent=Window;
  VbData->Enabled=TRUE;
  VBSetControlProperty (Control,Index_Property_ForeColor,
      GetInfoTextColor ());
  VBSetControlProperty (Control,Index_Property_BackColor,
      GetInfoBkColor ());
  VbData->AutoBrush=CreateSolidBrush (GetinfoBkColor());
  }
```

在这里，我们也把从用户调色板中得到的颜色值作为初始值赋给前景角和背景色。这些颜色不一定能够与终端用户的调色板一致，但是我们需要有初始值。

在删除 Tips 控件时，我们创建的刷子也得删除：

```
static void OnDestroy(HCTL Control,HWND Window)
  {
  LPVBDATA VbData-(LPVBDATA)VBDerefControl(Control);
  DeleteObject(VbData->AutoBrush);
  OnMethodClear(Control,Window);
  }
```

我们还要释放掉所有仍在内存里的标题 HSZ，这可以调用 OnMethodClear()完成。

如果用户选定了 AutoColor 属性，我们就需要响应消息 WM_SYSCOLORCHANGE 以重新创建刷子：

```
static void OnSysColorChange(HCTL Control)
  {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  if(IsWindow(VbData->TipWindow))
```

```
    DestroyWindow(VbData->TipWindow);
  DeleteObject(VbData->AutoBrush);
  VbData->AutoBrush=CreateSolidBrush(GetInfoBkColor());
  }
```

我们的提示窗口会发送 WM_CTLCOLOR 消息给 Tips 控件；由 Tips 决定该如何响应它：

```
static HBRUSH near pascal OnCtlColor(HCTL Control,HWND Window,HDC dc)
   {
  LPVBDATA VbData=(LPVBDATA)VBDerefControl(Control);
  HBRUSH Result;
  if(VbData->AutoColor)
   {
  SetTextColor(dc,GetInfoTextColor());
  Result=VbData->AutoBrush;
  }
else
  Result=(HBRUSH)SendMessage(GetParent(Window),
      WM_CTLCOLOR, (WPARAM)dc,
      MAKELONG(Window,CTLCOLOR_STATIC));
return Result;
}
```

在这个处理程序中，先检查属性 AutoColor 的当前值。如果此值为真，函数就自己对 WM_CTLCOLOR 消息作出响应，即设定正文颜色并返回一个刷子句柄。否则，就向本窗口的父窗口请求这个刷子，这就是在 VBX 中获得后备颜色属性的方法。

下面是控件过程自身。它有的是由 VBX Genie 给出的默认代码，有的是我们已经见过的函数的调用。只有一点例外：附加了响应 WM_SETFONT 和 WM_GETFONT 消息的代码：

```
        case WM_SETFONT:
            ((LPVBDATA) VBDerefControl (Control)) -> TipFont = (HFONT)
            wParam,
            break;
        case WM_GETFONT:
        return(LRESULT)(int)
        ((LPVBDATA)VBDerefControl(Control))->TipFont;
```

如果您的控件用到了任何标准的 Font 属性，那么必须要对这两个消息提供支持。通常的处理方法如您在这里所见到的：响应 WM_SETFONT 消息，字体被保存；响应 WM_GETFONT 消息，字体被返回。我们并不拥有这字体，它仍然属于父窗口，或任何发送给我们这个消息的窗口。所以我们不必在以后删除它。

有了对上述内容的理解，下面给出整个控件过程：

```
long far pascal _export CtlProc(HCTL Control,HWND Window,USHORT Msg,USHORT
wParam long lParam)
  {
```

```
switch(Msg)
    {
    case WM_CREATE:
        OnCreate(Control,Window);
        break;
    case WM_SYSCOLORCHANGE:
        OnSysColorChange(Control);
        break;
    case WM_CTLCOLOR:
        return(LRESULT)(int)(HBRUSH)
                    OnCtlColor(Control,Window,(HDC)wParam);
    case WM_SETFONT:
        ((LPVBDATA)VBDerefControl(Control)) ->TipFont=(HFONT)wParam;
        break;
    case WM_GETFONT:
        return(LRESULT)(int)
                ((LPVBDATA)VBDerefControl(Control)) ->TipFont;
    case WM_ENABLE:
        ((LPVBDATA)VBDerefControl(Control)) ->Enabled=wParam;
        break;
    case VBM_METHOD:
        return OnMethod(Control,Window,wParam(void far*)lParam);
    case VBM_SETPROPERTY:
        {
        LRESULT Result=0;
        if(OnSetProperty(Control,Window,wParam,lParam,&Result)) return Result;
        }
        break;
    case VBM_GETPROPERTY:
        {
        LRESULT Result=0;
        if(OnGetProperty(Control,Window,wParam,(LPVOID)lParam,&Result))re-
        turn Result;
        }
        break
    case VBM_GETPROPERTYHSZ:
        if (OnGetStringProperty (Control, Window, wParam, (HSZ far*)lParam)) re-
        turn 0;
        else
                break;
```

```
    case VBM_INITPROPPOPUP:
        switch(wParam)
                {
                case Index_Property_About: return(LRESULT)(int)PopupAbout
                ();
                }
        break;
    case VBM_HELP:
        if(OnHelp(Window, LOBYTE(wParam), HIBYTE(wParam), Properties,
        Events)) return0;
        break;
    case WM_DESTROY:
        WinHelp(Window,HelpFileName(),HELP_QUIT,0);
        OnDestroy(Control,Window);
        break;
            }
    reurn VBDefControlProc(Control,Window,Msg,wParam,lParam);
    }
```

MODEL 结构和 VBINITCC()、VBTERMCC()函数由 VBXGenie 生成，无需再作修改:

```
MODEL Model=
    {
    VB300_VERSION,
    MODEL_finvisAtRun,
    (PCTLPROC)CtlProc,
    0,
    0,
    sizeof(VBDATA),
    8000,
    "Tips,"
    "Tips",
    NULL,
    Properties,
    Events,
    0,
    0,
    (BYTE)-1,
    100
    };
  BOOL far pascal_export VBINITCC(USHORT Version,BOOL Runtime)
   {
```

```
    if(! Runtime)
        RegisterVbPopups();
    RegisterTipClass(LibInstance);
    return VBRegisterModel(LibInstance,&Model);
    }
VOID FAR PASCAL_export VBTERMCC(void)
    {
    UnregisterVbPopups()
    }
```

函数 Register TipClass()在 TIP. C 中实现，其原型在 INTERNAL. H 中；我们稍后要讲到它。

13.2.6 告诫与警示

请不要把 Tips 控件与提示窗口相混淆，Tips 控件是您放置在某个表上的控件，而提示窗口却是鼠标光标移动到按钮上时显示出的窗口。Tips 控件把工具窗口作为子类(在函数 OnMethodAddItem()的代码中您已见到这是如何实现的)，然后它们负责监视有无鼠标移过自己。这是在 OnMouseMove()事件处理程序中实现的：

```
static void near pascal Release Tip (LPVBDATA VbData,HWND Window);
#define PROBABLY_UNIQUE_TIMER_ID 10021
static void OnMouseMove (LPVBDATA VbData,
            HWND Window,POINT Point)
  {
  if(1VbData->MouseCaptured)
     {
     SetCapture(Window);
     SetTimer(Window,PROBABLY_UNIQUE_TIMER_ID,
           VbData->Delay,NULL);
     VbData->MouseCaptured=TRUE;
     }
  else
     {
     RECTRect;
     GetClientRect(Window,&Rect);
     if(1PtInRect(&Rect,Point))
          Release Tip(VbData,Window);
     }
  }
```

事件发生时，可能存在两种状态：鼠标已被俘获或鼠标还未被俘获。如果未俘获，就设置一个计时器。有一点很重要，就是这个计时器不能和子类控件拥有的任何计时器发生冲突。所以我随机选了一个数字并把它定义为 PROBABLY_UNIQUE_TIMER_ID；用这个数字确定

我们的计时器。如果您预料这会与某个控件相冲突，可换一个数字试试。

俘获鼠标的含义是，即使鼠标光标从子类控件上移开之后，我们还继续接收到来自它的消息。这样，如果鼠标已被俘获，我们就查看其坐标是否仍落在矩形范围内。如果鼠标未被俘获，就调用 ReleaseTip()，它或者终止计时器，或者撤消现有的提示窗口。

如果用户让鼠标光标在子类控件上停留的时间超过了 VbData->Delay 所规定的毫秒数，计时器就会发送一个消息给它。这个消息是由 OnTimer()函数处理的：

```
static void near pascal OnTimer(LPVBDATA VbData,HWND Window,int i)
 {
 static BOOL Busy=FALSE;
 RECT Rect;
 if(!Busy)
   {
   Busy=TRUE;
   KillTimer(Window,PROBABLY_UNIQUE_TIMER_ID);
   if(IsWindow(VbData->TipWindow);
       DestroyWindow(VbData->TipWindow);
   if(VbData->Enabled)
     {
     GetWindowRect(Window,&Rect);
     VbData->TipWindow=CreateWindow("Tip",
       VBDerefHsz(VbData->Tools[i].Caption),
       WS_POPUP|SS_SIMPLE,
       Rect. right-10,
       Rect. bottom-10,
       50, 50
       VbData->TipParent,
       0,
       LibInstance,
       (LPVOID)VbData->TipParent);
     SendMessage(VbData->TipWindow,
       WM_SETFONT,(WPARAM)VbData->TipFont, FALSE);
     ShowWindow(VbData->TipWindow,SW_SHOWNOACTIVATE);
     }
   Busy=FALSE;
   }
 }
```

在这个函数里，我们所第一关心的一件事情是，在处理前一个 WM_TIMER 消息期间不会被后一个 WM_TIMER 消息再次触发。static 变量 Busy 负责预防这种情况的发生。然后就马上终止计时器。

如果碰巧有一个提示窗口正显示着，我们就撤消它。按理说不会发生这种情况；但还是小

心为妙。

只要控件目前是激活的，我们接下来就要为当前工具创建提示窗口了。请注意没有设定 WS_VISIBLE 标志，因为我们下面还要调用 ShowWindow()。同样还要注意，我们是把提示控件自身的句柄作为最后一个参数来传递的。这样做是为了迁就 Windows 中的一个奇怪现象，我以前还未曾注意到。

CreateWindows()的第八个参数应该是控件的父窗口句柄。正如我们所期望的那样，它通常都是可以正常工作的。然而，Tips 控件是子窗口，而提示窗口自己又是弹出式窗口。显然，Windows(或者说，至少 Windows95)查找弹出式窗口的父窗口的方法是，从实际的父窗口出发，反复调用 GetParent()，找到第一个弹出式窗口或重叠式窗口，并自动地把它认定为该弹出式窗口的父窗口。由于我们需要让提示窗口发送 WM_CTLCOLOR 消息给 Tips 控件(通常都是发送给父窗口)，因此我们必须换一种方式给出 Tips 控件的窗口句柄，我把它作为 CreateWindow()的参数。

创建了窗口之后，我们给它发送一个 WM_SETFONT 消息，把 Tips 控件通过同类消息(WM_GETFONT)从 VBX 管理器获得的字体传递过去。

最后，调用 ShowWindow()显示出新的提示窗口，SW_SHOWNOACTIVE 风格保证了此前拥有焦点的窗口不会失去焦点。

我们已经见到了，WM_MOUSEMOVE 消息可能导致提示窗口被撤消掉，或计时器被终止。这些都是在 ReleaseTip()函数中实现的：

```
static void near pascal ReleaseTip(LPVBDATAT VbData,HWND Window)
  {
  ReleaseCapture();
  KillTimer(Window,PROBABLY_UNIQUE_TIMER_ID();
  VbData->MouseCaptured=FALSE;
  if(VbData->TipWindow)
      {
      DestroyWindow(VbData->TipWindow);
      VbData->TipWindow=NULL;
      }
  }
```

如果计时器已被终止(在提示窗口被创建之初就应该是这样)，KillTimer()函数就会调用失败；但是这个失败不会引起大的动作，所以用这个函数来做一般的清理工作是没什么害处的。

剩下的只是子类窗口过程了，它把上面所讲的内容都串接起来：

```
LRESULT_expolt FAR PASCAL ToolWndProc(HWND Window,UINT Msg,
                    WPARAM wParam,LPARAM lParam)
  {
  HCTL Control=
    (HCTL)MAKELONG(GetProp(Window,"TipControlL"),
      GetProp(Window,"TipControlH"));
  LPVBDATA VbData-VBDerefControl(Control);
```

```
    register i=(int)GetProp(Window,"TipIndex");
    switch(Msg)
      {
     case WM_MOUSEMOVE:
         OnMouseMove(VbData,Window,MAKEPOINT(lParam));
         break;
     case WM_KILLFOCUS:
     case WM_DESTROY:
         if((HWND)lParaml=VbData->TipWindow)
           ReleaseTip(VbData,Window);
         break;
     case WM_TIMER:
                 OnTimer (VbData, Window, i);
                 break;
               }
               return CallWindowProc(VbData->Tools[i].WndProc,
         Window,Msg,wParam,lParam);
     }
```

在这里，您见识了 Control 句柄的两半是如何重新组装起来的，以及在最后如何调用工具的原始窗口过程(通过 CallWindowProc())以取代 DefWinowProc()。即使是我们已经处理过的消息，也应调用原始窗口过程，以确保工具的原有表现不会发生改变。

13.3 提示窗口

基于重用性的考虑，我把提示窗口单独做成一个代码模块。提示窗口相当简洁，实际上，我是以创建一个简单的 STATIC 弹出式窗口来开始我的 Tips 控件的开发工作的。我想要窗口其大小应该能够作自动调整，这个特点却不是 STATIC 类所具有的；因此我只得依靠自己的力量了。“Tip” 类是在 TIP. C 中实现的。

事实上，就象所有的窗口那样，提示窗口也必须维护某些局部信息。我把这些信息存储在局部结构 MyData 中，该结构与控件过程的 VbData 结构类似：

```
xypedef struct
  {
  HFONT Font;
  HWND Tool;
  } MYDATA, far*LPMYDATA;
```

按标准做法，结构是在处理 WM_NCCREATE 消息的时候分配的：

```
static BOOL near pascal OnNcCreate(HWND Window,LPCREATESTRUCT cs)
  {
  BOOL Result=(BOOL)DefWindowProc(Window,
     WM_NCCREATE,0,(LPARAM)cs);
```

```
LPMYDATA MyData;
if(!Result)
    return Result;
MyData=malloc(sizeof MyData);
if(MyData)
    {
    memset(MyData,0,sizeof MyData);
    MyData->Tool=(HWND)(long)cs->lpCreateParams;
    if(1 MyData->Tool)
        MyData->Tool=cs->hwndparent;
    SetWindowLong(Window,0,(long)MyData);
    return TRUE;
    }
else
    return FALSE;
}
```

在做其他事情之前，务必先调用 WM_NCCREATE 消息的处理器，这一点绝对很重要。如果没有足够的内存用来创建窗口，这个消息的处理可能会失败。类似地，如果没有足够的闲置内存用于局部结构，那么必须返回 FALSE 而不是 TRUE，以此表明本消息的响应已失败。

在窗口创建之后不久，我们应该会接收到消息 WM_SETFONT。除了保存字体句柄，我们还必须检测出用新字体显示窗口标题所需的空间大小：

```
static void near pascal OnSetFont(LPMYDATA MyData,HWND Window,HFONT Font)
  {
  RECT Rect;
  HDC dc=GetDC(Window);
  HFONT OldFont=NULL;
  char Buffer [128];
  DWORD Extent;
  MyData->Font=Font;
  if(MyData->Font)
      OldFont-SelectObject(dc,MyData->Font);
  GetWindowText(Window,Buffer,sizeof Buffer);
  Extent=GetTextExtent(dc,Buffer,strlen(Buffer);
  GetWindow Reet (Window,&Rect);
  MoveWindow(Window,Rect.left,Rect.top,
      LOWORD(Extent)+4, HIWORD (Extent) +4, FALSE);
  if(OldFont)SelectObject(dc,OldFont);
  ReleaseDC(Window,dc);
  }
```

尽管对 VB 控件不会有这类事情发生，但是一个窗口应该对 WM_SETFONT 消息有所准

备，接受由它指定的系统字体。即，在 VB 控件的 WM_SETFONT 消息中，包含有新字体句柄的参数实际上是 NULL。对这个情况的处理是，假定已经给出实在的字体句柄，并据此改变设备环境的字体。

在确定了标题所占空间大小(通过调用 GetTextExtent ()) 之后，把各边都稍微扩大一点以留出空来，并调用 MoveWindow()相应调整窗口大小。

绘制提示窗口只有一点值得一提，为了获得背景色和前景色，必须发送 WM_CTLCOLOR 消息，以允许改变文本颜色及提供背景刷子。通常这个消息都是发送给父窗口；但是在这里它却必须发送给 Tips 控件：

```
static void near pascal OnPaint(LPMYDATA MyData,HWND Window)
 {
  PAINTSTRUCT ps;
  HDC dc=BeginPaint(Window,&ps);
  char Buffer [128];
  RECT Rect;
  HBRUSH Brush, OldBrush=NULL;
  HFONT OldFont=NULL;
  if(MyData->Font)
      OldFont=SelectObject(dc,MyData->Font);
  GetWindowText(Window,Buffer,sizeofBuffer);
  GetClientRect(Window,&Rect);
  Brush=(HBRUSH)SendMessage(MyData->Tool,
      WM_CTLCOLOR,(WPARAM)dc,
      MAKELONG((WORD)Window, CTLCOLOR_STATIC));
  if(Brush)OldBrush=SelectObject(dc,Brush);
  Rectangle(dc,Rect. left,Rect. top,Rect. right,Rect. bottom);
  SetBkMode (dc,TRANSPARENT);
  TextOut(dc,2,2,Buffer,strlen(Buffer));
  if(OldFont)SelectObject(dc,OldFont);
  if(OldBrush)SelectObject(dc,OldBrush);
  EndPaint(Window,&ps);
  }
```

如果提供了新的刷子，我们就把它选入设备环境。然后绘制窗口的边框和文本，这以后，再重新选入原始字体和刷子。

提示窗口过程没有什么惊人之处：

```
LRESULT_ exoport FAR PASCAL TipWndProc (HWND Window, UINT Msg, WPARAM wParam,
                                        LPARAM lParam)
 {
 LPMYDATA MyData=(LPMYDATA)GetWindowLong(Window,0);
 LRESULT Result=0;
```

```
switch(Msg)
  {
  case WM_NCCREATE:
      Result=OnNcCreate(Window,(LPCREATESTRUCT)lParam);
      break;
  case WM_NCDESTROY:
      free(MyData);
      break;
  case WM_SETFONT:
      OnSetFont(MyData,Window,(HFONT)wParam);
      break;
  case WM_GETFONT:
      Result=(LRESULT)(int)MyData->Font;
      break;
  case WM_SYSCOLORCHANGE:
      Invalidate Rect(Window,NULL,TRUE);
      break;
  case WM_PAINT:
      OnPaint(MyData,Window);
      break;
  case WM_SETFOCUS:
      SetFocus((HWND)) wParam);
      break;
default:
    Result=DefWindowProc(Window,Msg,wParam,lParam);
  }
return Result;
}
```

当窗口撤销时，MyData被释放；所有响应WM_SETFONT消息得到的字体，在响应WM_GETFONT的询问时被返回；终端用户如果改变了系统颜色，就令窗口重新绘制自己。此外，提示窗口不允许把焦点设置到它上面，如果试图这样做，它就把焦点又设置到失去焦点的窗口。

类的注册包含在TIP. C当中：

```
static WNDCLASS WndClass-
  {
  0,
  TipWndProc,
  0,
  sizeof(LPMYDATA),
  NULL,
```

```
NULL,
NULL,
NULL,
NULL,
" Tip"
};
void far pascal Register TipClass(HINSTANCE Instance)
  {
  WndClass. hCursor=LoadCursor(NULL,IDC_ARROW);
  WndClass. hInstance-Instance;
  if(1RegisterClass(&WndClass))
      MessageBox(NULL,"Registration of Tip class failedl",
        "Tip",MB_ICONSTOP);
  }
```

函数 Register TipClass()原型在 INTERNAL. H 中;在 VBINTCC()函数中我们已经见到了对它的调用。

13.4 应用 Tips 控件

要应用 Tips 控件,只需简单地把这个 VBX 加入到您的定制控件列表中,如果您想要给一个表上的某个控件添上有关的用户提示,就可以放置一个 Tips 控件,设置好 Deley 属性(可能还要包括 ForeColor、BackColor 或 AutoColor 属性)。然后,在表的 Load 事件处理程序中,以相关工具的窗口句柄为索引调用 AddItem 方法,为控件加入提示信息。例如:

```
Sub Form_Load()
Tipsl. AddItem" Cut", cmd EditCut. hWnd
Tipsl. AddItem" Copy", cmd_EditCopy. hWnd
Tipsl. AddItem" Paste", cmd_EditPaste. hWnd
End Sub
```

如果愿意,您还可以在任何时候改变标题的文本内容,同样这也需要以工具窗口句柄为索引:

```
Tipsl. Captions(cmd_EditCut.hWnd)=" Cut selected circuit"
```

要做的就是这些。您没有理由拒绝在应用程序里使用提示工具,即使编写的是16位版本的应用程序。

第四部分 Microsoft 的 OLE Control Wizard

第十四章 什么是 COM

在第二章，我总体介绍了 OCX：它们是为谁服务的、在高层次上它们是如何工作的，至少它们是做什么的等等。但实际回顾一下，我花了更多幅篇来讲它们不能做什么，而不是它们能做什么。其实那是必要的，是对 Microsoft 公司的市场部为了促销在这个概念上所做工作的一点简单的补充。

这里我们就要真的来学习 OCX 了，并且我们还要编写一些 OCX；所以我们最好要了解一些细节，获得一些背景知识，这样你就可以清楚地知道当你的 OCX 运行时在 MFC 内部会有什么操作。

14.1 介绍公共对象模型

当 Microsoft 公司首次提出 COM 思想时，COM 是指“Component Object Model(组件对象模型)”。后来 Microsoft 发现有很多其它公司参与进来了，所以 COM 被更名为“Common Object Model(公共对象模型)”。

公共对象模型的目标是使所有的计算机软件组件都有相同的程序接口(至少在底层是这样的)。举例说明：在面向对象的语言(如 SmallTalk)中，所有的对象类都派生于同一个基类“Object”。从 COM 观点出发，所有的软件组件都派生于同一个叫做 COM 的基“类”。

COM 被设计成“二进制标准”。这就是说，它试图成为独立于语言的。事实上，我读过的所有关于 OLE 的书中都是用 C 和 C++例子来呈现 COM 的，好象 C 和 C++是唯一的计算机语言似的。实际上，COM 要求具有可以使用任何一种语言来实现 COM 对象的特点，特别是要能通过指针来调用函数。

COM 对象是通过接口(interface,即几组函数)来获取的。虽然接口应该设计成独立于语言的，但它们被设计成好象是带有虚函数的 C++类了。因此，要用 C 来实现这些接口，你需要手工编写作为函数指针结构的虚方法表。不用多说，很少有程序员实际上用 C 去编写 COM 对象程序。

在 C++中，每个接口都被设计成好象是个 C++类。有一点很重要：接口并不是类的对象。所以 Microsoft 对接口类习惯使用 I 而不是 C 前缀，如 IUnknown。

IUnknown 是每个 COM 对象都必须实现的一个接口，否则那个对象就不是个 COM 对象。从 C++类的角度来说，其它接口都派生于 IUnknown。所以，我们最好从仔细讨论 IUnknown 接口开始。

14.2 介绍 IUnknown 接口

要了解 COM 接口是如何工作的，我们需要做的第一件事情是看看它的说明。但要找到 I-Unknown 的说明就好象要到行李船舱底部去找你的钱包一样。当你看到 VisualC++4.0中包含的一组头文件时，你会发现一个叫做 OLE2.H 的头文件。它看起来好象是个好的出发点，是吗?不幸的是，IUnknown 接口的说明不在那儿。另一个可能的头文件是 COMPOBJ.H，因为 COM 原来的意义是组件对象模型。但这个文件所提供的唯一线索是预编译消息："WARNING：your code should #include objbase.h instead of compobj.h"。

OBJBASE.H 头文件使我们非常接近 IUnknown 了，因为这个文件中有一些宏使用了 IUnknown。所以我们必须进一步寻找。最终你会发现我让你找来找去的目的是让你明白 OLE2.0确实不是一种你在一周内就可以掌握的技术，这并不是因为它本身是如何复杂，而是因为它被设计得过头了。

实际上，IUnknown 的结构如下：

```
class IUnknown
  {
 public:
    virtual long QueryInterface(long riid,void far*far ppvObject)=0;
    virtual long AddRef(void)=0;
    virtual long Release(void)=0;
 }
```

你要明白，我已经很大程度地简化了这个结构。首先，函数的参数并不是 long，而是通过 typedef 定义的相当于长整型的类型。ppvObject 参数的双 far 指针又是怎么回事呢?far 可能是被_RPC_FAR 宏或可能是 typedef 语句隐藏起来了，我也没能找到它的定义。再进一步说，你实际上是不会找到 "class IUnknown" 语句的。那是因为 "class(类)" 已经被类型定义(typedef)成 "interface" 了，这意味着它是一个 "结构"。所以，下面才是头文件中 IUnknown 真正的说明代码：

```
interface IUnknown
 {
public:
      virtual HRESULT_stdcall QueryInterface(
        /*[in]*/REFIID riid,
        /*[out]*/void__RPC_FAR*__RPC_FAR*ppvObject)=0;
      virtual ULONG_stdcall AddRef(void)=0;
      virtual ULONG_stdcall Release(void)=0;
};
```

作为 COM 对象的使用者，你不用关心它是如何实现的。但如果你要编写一个 COM 对象，你就必须提供它的实现。所以，我们来看看那三个函数是做什么的。

首先要注意，它并没有构造函数。那是因为接口并不是一个对象，它的构造函数是 C++ 的语法所不允许的。你必须用其它方式来创建 COM 对象。一般是调用 CoCreateInstance()函

数。你必须为这个函数提供一个叫做全局唯一标识符(GUID,globally unique identifier)的参数,CoCreateInstance()函数会给你返回一个指针,它不是指向某个对象而是指向 IUnknown 接口

实际上你根本不能获得某个 COM 对象本身。那是因为你无法知道对象的位置。它可以在你的计算机中某个单独的内存块中;它也可以在另一台计算机上;它也可以在另外一个星球上。这都无所谓。你可以通过它的接口做任何你需要的事情。

你要做的第一个事情是通知那个对象你已经有了它的接口了。你需要调用 AddRef()来完成这个任务。这个操作会增加内部的引用次数。我一直没搞清楚为什么这个操作不能在获得接口指针时完成。从面向对象的角度看,这样做似乎更纯一些。任何时候当你调用 AddRef()后,你最后必须调用 Release()。当 Release()的调用次数与 AddRef()的调用次数相等时,对象就可以删除自己或释放系统资源了。这种技术使单独一个对象就能给多个应用程序或同一应用的多个部分提供接口和服务,而那些对象的使用者无需相互了解对方。

你通过对象的 IUnknown 接口告诉对象你在使用它,而后你就可以使用它了。这样做固然很好,但这种功能并不是特别有用。所以 IUnknown 又提供另一个函数:QueryInterface()。你给它一个 REFIID,如果接口被这个 COM 对象所支持的话,你将获得一个指向它的指针。接下来,看在上帝的份上,不要忘记对这个新指针调用 AddRef()和最后的 Release()函数!

由于接口并不是对象本身,所以在你获得对象的接口(即使是 IUnknown)之前,你需要用一种方式来创建一个对象。上面提到过,一种方法就是调用全局函数 CoCreateInstance()。一旦你有了一个现成的对象,你通常可以使用 IClassFactory 接口来生成另一个同类型的对象。这个接口也被 CoCreateInstance()所使用,所以很显然,有可能在创建对象之前就可以获取到这个接口。至于如何获取,方法还没有被公开。无论如何,所有的 COM 对象都必须同时支持 IClassFactory 接口和 IUnknown 接口。

我在第二章提到过,COM 没有定义用来枚举可用接口的方法。相反,你必须向它一个一个地询问是否支持你想要的接口。其主要思想是只提供小数量的每个人都可能使用的接口;但据我所知,还没有人曾公开过这种接口列表,告诉我们都有哪些接口可用。所以创建新 COM 对象的人都倾向于引入新的接口。同时,当某一个接口被证明不够用的时候,就会有新的接口代替它。并且没有一种机制来规定接口的版本。所以我们现在有了 IClassFactory 接口和它的增强版本 IClassFactory2。如果你需要进一步增强接口功能而编写 IClassFactory3,而同时另一公司的人也在做同样的事,那会怎么样?!

14.3 OLE 自动化的出现

看来在 COM 发展的早期,接口的泛滥一定是难以控制的,因为在 OLE2出现不久,紧接着又出现了 OLE 自动化(OLE Automation)。

OLE2和 OLE1允许一种叫做“服务器(server)”的特殊类型的应用程序支持“子文档(subdocument)”。如果在你的应用程序中字处理功能只是一个小的方面,你就不必编写完整的字处理器了。并且用户当然也愿意能够将,比如说,传单软件(spreadsheet)的一部分拖到一个字处理文档(比如,一个商业计划)中。但服务器程序经常使用复杂的宏语言,并且出现了一批新类型的程序员,他们的主攻目标就是以这些服务器作为开发平台来开发高度专业化的应用程序。

但 OLE 不能允许应用程序的一个部分编程控制另一部分。这就是推出 OLE 自动化的原

因。显然，OLE 自动化满足了编程控制 OLE 对象的需要，但它的实现也提供了解决接口问题的另一条途径。OLE 自动化并不是为每组函数生成新的接口，而是使用一个接口 IDispatch 来一劳永逸地解决各种问题。IDispatch 允许获取类似 VBX 的属性和方法，并向使用者提供相似于事件的反馈机制。这些属性、方法和事件可以通过另一个接口 ITypeInfo 用名字进行枚举和引用。通常是使用 ITypeInfo 来枚举所有的属性、事件和方法的名字，然后将这些名字传递给 IDispatch 接口，该接口将它们翻译成 DISPID(一组32位的数)。如果要重复使用这些名字，用 DISPID 来查找要比用字符串查找容易得多。

14.4 OLE 控件

作为 OLE 服务器的一个特色，OLE 自动化支持属性和方法。这样，一个传单(spreadsheet)可以包含有一个插入的 OLE 图形对象，而传单可以通过它的宏语言来编程控制这个 OLE 对象。

但如果用户改变了图源或在传单中使用就地编辑的功能(OLE2的另一特色)改变了图形的话该怎么办?或换句说，如果传单应用程序需要知道图形是否被改变的话，该怎么办?OLE2除了定期检查对象的属性外，没有提供其它任何方法来使你的容器知道这些。当你的容器需要知道嵌入的对象发生了什么事件时，OLE 服务器的功能是不够的，你需要的是 OLE 控件。

你应该知道，OLE 允许服务器被实现成既可以单独运行(这种情况下，它运行在自己的地址空间，也可能运行在另外一台计算机上)也可以成为就地服务器。就地服务器是一种 DLL，它运行在它的容器的地址空间中。请注意容器本身并不知道，也不应关心它所使用的服务器是单独运行的还是就地服务器。(这样就允许用单独执行的服务器创建一个嵌入对象，而用就地服务器来观看或播放该对象。)

OLE 控件通常被编写成就地运行方式。所以它从不会单独运行。特别是它并不象 OLE 服务器那样雄心勃勃。一个典型的 OLE 服务器可能是包括制表、编写报告、文件管理和打印等功能的传单应用软件(spreadsheet application)，而一个典型的 OLE 控件可能是作为传单背景的栅格控件，控件本身还远远不能称为传单。

有一种特殊类型的 COM 对象叫做集合(collection)。有了 OLE 集合，就可以象存取数组元素一样来存取控件(或它的数据)的多个实例。OLE 集合对象在它的核心部分支持加、定位、存取和删除它所包含的对象等操作。这些特色是通过 IEnumVARIANT 接口来提供的。我在这里提它是因为许多 OLE 控件(比如，那些基于 Windows 列表框的控件)可以实现成集合。

14.5 内存上的考虑

Microsoft 公司在疯狂地劝诱我们转向32位编程时忽略了一件事，那就是 Windows 95和 Windows NT 对32位内存空间的使用效率远不如 Windows 3.1对16位内存空间的使用。当运行16位应用程序时，Windows 95要在单独的地址空间中模拟 Windows 3.1运行方式：第二个应用程序实例使用第一个实例的代码段。这样，不管有多少复本在运行，应用程序的代码只被装入一次。这意味着第二个应用程序的实例要比第一个启动得要快。同样，16位的 DLL 也只被装入一次，并为所有应用程序所共享。

在32位线性地址空间中就不是这样了。每个32位的应用程序都在它自己的空间中运行。有

可能一个应用程序需要一个共享的内存块，但应用程序加载器并不会提供任何帮助。好的一面是如果应用程序的一个复本代码崩溃了，它并不会影响同时运行的同一应用程序的其它复本。坏的一面是每个复本的代码都要占用内存，因而就需要更多的内存，并且每个复本的装入都需要同样多的时间。实际上，由于申请更多内存而引起的页交换会使第二个复本的装入更慢！这就是为什么 Windows 3.1只需要8M 内存就运行得很好，而 Windows 95的运行则需要甚至两倍以上内存才可容忍的主要原因。

OLE 控件的 DLL(通常它有一个 OCX 后缀)一般要比相应 VBX 小一些起。但这有一点欺骗性。为了能运行，OCX 需要至少六个 OLE DLL 再加上 MFC DLL 的支持。这样由 OLE 容器所使用的第一个 OCX 要占用大量的硬盘和分布盘空间，并且安装和加载应用程序也很费时。如果你的应用程序运行在32位并且允许同时运行多个复本的话，问题将会更加严重。因为每个 DLL 必须被分别重复装入到各个复本的地址空间中。

14.6 属性类型

VBX 支持两类属性：标准的和用户的。标准属性是预实现的；你所要做的就是告诉它你需要哪一个。当然，用户属性的名字、存贮特点和实现则完全依赖于你的设计。

OCX 扩展了这个思想，而又增加了两类属性。除了用户属性和标准属性(在 OLE 中称为库 存属性)外，OCX 中还有环境属性和扩展的属性。

对用户属性的设置与在 VBX 中一样：你必须规定属性名和数据类型。当用 OLEControl Wizard(这是唯一的正规方法)创建控件时，你还要选择是自动存贮属性值，还是由你来实现设置和获取属性值的函数。

14.6.1 库存属性(Stock Property)

库存属性的列表随着 OLEControl Wizard 的版本不同而稍有差别。表14.1显示了一个常见的库存属性列表。

表14.1　　　　常见的 OCX 库存属性

库存属性	描　　述
BackColor, ForeColor	它们分别定义了控件背景颜色以及文本和线的颜色。与 VBX 中相应的属性相似。
BackStyle	定义了控件中的文本和点线以及阴影线是如何显示的：透明(Transparent, 1)或不透明(Opaque, 0)。与 VBX 中的同名属性相似。
Caption, Text	规定了控件的标题或内容的文本。这两个属性是互斥的。它们与 VBX 中同名属性功能相同。
Enabled	规定控件是否可以接收键盘输入。与 VBX 中同名属性功能相同。
Font	定义了在控件内所显示的文本的字体。这个属性代替了全部的 VBX 字体属性：FontBold、FontIlalic, FontName, FonStrike 和 FontUnderline。

14.6.2 扩展的属性(extended property)

如果你在表14.1中没见到你喜欢的标准属性，你也不要奇怪。一些属性在 VBX 中是标准属性，而在 OCX 中被放到了单独的一个类中了。这类属性叫做扩展的属性。扩展的属性区别于库存属性的关键所在是这些属性是容器所需要的并且是由容器而不是由控件来管理的。实际

上，虽然控件可以获取这些属性的值，但它几乎从不需要这样做。扩展的属性列表显示在表14.2中。

表14.2 **组 OCX 扩展的属性**

扩展的属性	描述
Cancel、Default	应用于按钮类型的控件。设置按钮如果 Esc 键（Cancel）或回车（Default）键被按下时会自动被点击。这个属性与 VBX 中的同名属性的功能一样。
Name	规定了容器所提供的对象的名字。这个属性与 VBX 中的同名属性的功能一样。
Parent	标识控件所嵌入的文档。控件可以使用这个属性得到它的容器。
Visible	如果控件是可见的，这个属性为 TRUE。这个属性与 VBX 中的同名属性的功能一样。

14.6.3 环境属性(Ambient Property)

最令人兴奋的新属性是环境属性。控件有时可能想改变这些属性的值，但大多数情况下是从容器"继承"这些属性的值。比如，你一定有很多次在 Visual Basic 中的表(form)上(如 About 对话框)放置一组标签控件并改变表的颜色后，要精心改变每个标签控件的 BackColor 属性的值来进行颜色匹配。如果使用 OCX 控件，环境属性 BackColor 可以用于为库存属性 Backlolor 提供一个数值。如果你愿意，你可以忽略那个库存属性；因为控件的背景颜色最终还是要根据容器的背景颜色来调整。

表14.3列出了环境属性。其中，许多属性的名字与库存属性是相同的。

表14.3 **一组 OCX 环境属性**

环境属性	描　　述
DisplayAsDefaultButton	按钮类控件要读取这个属性值以确定它是显示成默认按钮。（它应该可以通过读取自己的扩展属性 Default 来获得相同的信息。）
LocaleID	通过这个属性控件可以知道它所嵌入的应用程序运行。
MessageReflect	如果这个属性值为 TRUE、那么由控件发送给容器的消息应该自动地反送回来。这与 VBX 有点相似：当 VBX 管理器接收到控件的 WM＿COMMAND 通知消息后，要向控件过程发送 VBW＿COMMAND 消息。
ScaleUnits ShowGrabHandles	这个属性规定了由容器所使用的坐标系统名，如"Twips" 抓握手柄（Grab handle）是当控件被移动或改变大小时所显示的小东西。 如果这个属性值为 TRUE 并且 UIDead 为 FALSE，控件应该显示抓握手柄。
SupportsMnemonics	如果是 TRUE，容器则支持标题中的热键以允许用键盘快速存取控件。
TextAlign	文本对齐方式：0是默认值（数字是右对齐、文本是左对齐）；1是左对齐；2是居中；3是右对齐；4是两边同时对齐。注意，这些数并不与许多 VBX 通常使用的 TextAlign 属性相对应。
UIDead	一般情况下，如果容器的 Enabled 属性为 FALSE，这个值将为 TRUE。这允许容器在被禁止时自动禁止所有它包含的控件。但我有一点不明白，它为什么不调用 Enabled 属性呢？
UserMode	如果控件被终端用户所使用时，这个属性返回 TRUE；如果处于设计模式时，它返回 FALSE。

要获取到环境属性，OLE 控件必须获得它的容器的一个接口。因为环境属性实际上是容器的属性而不是控件自身的属性！这意味着容器本身必须是一个 COM 对象。这就是为什么 VB4.0是一个 OLE 应用软件的原因。

14.7 库存方法

VBX 有很多可用的库存方法。OCX 所提供的则是很少的，只有两个：DoClick 和 Refresh。DoClick 模拟用户在控件上的鼠标点击操作。Refresh 强制控件重画自身。

如果你想用 Control Wizard 实现类似集合的控件，你必须自己加入 AddItem、RemoveItem 以及类似的方法。或者你从头编写控件代码，通过实现 IEnumVARIANT 接口使它成为 OLE 容器。

14.8 事件

与定义四种属性一样。OLE 控件也定义了四种事件。它们是 Request(询问)类事件、Before(之前)类事件、After(之后)类事件和 Do(做)类事件。但与那四种属性不同的是，除了其中两类事件需要单个参数外，这些事件之间并没有本质的区别。

一个可取消的操作发生之前，Request 类事件会被触发。其效果是容器有机会取消这个操作。Request 类事件能支持任意多的参数，但最后一个参数 Cancel 是布尔型的引用。当这个事件被触发时，这个参数一开始被设置为 FALSE；但如果不希望发生这个事件所警告的操作的话，事件处理器要将它设置为 TRUE。我们在对这类事件命名时加入“Request”这个词，如 RequestUnload。在处理外部事件之前，Before 类事件会被触发。例如，标准 Windows 编辑控件(edit control)每次在控件内容要发生改变时会生成 EN_UPDATE 通知消息。如果你打算用这个通知消息来触发一个事件，你可以称它为 BeforeChange 的事件。

同样，After 类事件是在处理操作已经发生后被触发的。所以它不仅仅是一个通知。标准 Windows 编辑控件所发送的 EN_CHANGE 通知消息就是一个例子。如果你打算用这个通知消息来触发一个事件，你可以称它为 AfterChange 事件。

Do 类事件是指那些处理过程已经被实现了，但用户或程序员可能会增强或重新设计实现的事件。这类事件的最后一个参数通常是 EnableDefault，它是个布尔型参数。当这个事件被触发时，这个参数值被设置成 TRUE。如果你想用自己的代码替换掉原有的实现，除了要做一些其它的工作外，还要在触发这个事件时设置 EnableDefault 为 FALSE。

Microsoft 公司鼓励我们将事件类型名做为事件名的一部分：比如，RequestChange、BeforeChange、AfterChange 以及 DoChange。我赞成这个建议。但 Microsoft 公司提供给我们的8个库存事件并没有按照这种约定进行命名。表14.4列出了那些库存事件。

表14.4　　OCX 的库存事件

库存事件	描　述
Click，DblClick	当用户在控件上点击鼠标时，这个事件被触发。有些控件的 Click 事件被用来表明它的值已经改变了。
Error	当你的控件中有异步的、外部的错误发生时，这个事件被触发。这个事件包含了足够多的参数以便使你能清楚地告诉容器到底出了什么错。
KeyDown，KeyUp	当用户按下或释放某一键并正当控件具有输入焦点时，这些事件被触发。键的代码是通过一个引用类型的参数传递的：如果你不想库存事件对这个键进行处理，你可以将这个参数设置为0.
MouseDown，MouseMove，MouseUp	除了点击外，控件上的其它鼠标操作会触发相应事件。与 Click 事件不同，这些事件包含有鼠标坐标信息。

14.9 VBX 和 OCX 的比较

毫无疑问，OCX 的功能要比 VBX 强得多。问题是为获得这份增强的功能，我们要付出多大的代价？我开始着手测试 VBX、16位 OCX 和32位 OCX 的相对速度。我对三个方面感兴趣：相应的应用程序的加载时间、控件属性的存取时间和重画时间。

首先，我用 C 编写了一个简单的程序加载器。它先记录当前时间，而后加载应用程序（程序加载器的命令行参数中规定了应用程序名），再计算并显示所花时间：

```
#include<windows. h>
int PASCAL WinMain(HINSTANCE Instance,HINSTANCE Previous,LPSTR Command,int CmdShow)
{
  char Buffer [128];
  DWORD StartTime=GetTickCount();
  WinExec(Command,CmdShow);
  wsprintf(Buffer,"%s took %ld milliseconds to start",Command,GetTickCount()-StartTime);
  MessageBox(NULL,Buffer,"Timed Launch",MB_ICONINFORMATION);
  return();
}
```

我是以16位代码来编译这个程序的。这个程序叫 Launch。为运行它，我只需把一个应用程序拖到文件夹(Folder)中或资源管理器(Explorer)中的这个程序加载器的程序名上。

接下来，我以 VBX、16位 OCX 和32位 OCX 的形式设计了三个功能相同的简单的用户控件。这些控件用白色绘制背景，并在上面画一个彩色的椭圆。它只有五个属性：Left、Top、Width、Height 和 TestMe。属性 TestMe 是用来测试属性设置速度的。

当然，我是用 VBX Genie 来生成 VBX，而用 Control Wizard 来生成两种 OCX 的。为了生成16位的 OCX，我必须使用 VisualC++1.52。更早期版本 VC 所带的 Control Wizard 所生成的 OCX 不能与 Visual Basic 兼容。我只是简单地运行 Control Wizard、接受默认设置并生成 OCX 框架。接着我修改 OnDraw()函数使之与生成的 VBX 相一致，并且加入 TestMe 属性。(OCX 内部本身含有大小和位置属性。)

我所建的工程使用了发行版的库，选择了优化设置。这样所生成的控件与真正所要发行的控件没什么两样。我告诉编译器生成386或更高档 CPU 的指令代码，因为 Windows95或 VisualBasic40只能在386以上档次的机器上运行。

理论上讲，你可以用32位编译器简单重新编译你的16位 OCX 代码就可以得到32位的 OCX。但当我使用 VisualC++4.0进行编译时，编译旧版本的 Control Wizard 代码引发了一个链接错。我没有试图排除这个错，而是使用新版 Control Wizard 来生成相应 OCX，只是从16位 OCX 中复制了作画代码。从编译到连接共花15分多钟。(或许我应告诉大家，我是在8M 内存的386—40的机器上运行 Windows95的。尽管如此，我还是觉得生成得慢了一些。)

编译完三个控件后，我准备进行第一项比较：文件大小的比较。所有这三个控件都是发行版、面向80386的 CPU、优化了运行速度的。表14.5列出了这三个文件的大小。

表14.5　　三个控件文件大小的比较

控件类型	文件大小
VBX	31.9K
16位 OCX	21.4K
32位 OCX	23.0K

发行 VBX 会轻松一些，因为发行 OCX 必须还要附带上好几个 OLE 和 MFC 的动态连接库。

接下来我创建了三个 Visual Basic4.0工程。这三个工程除了放在表 Forml 上的控件不同外，其它方面都一样。每个工程都有两小块代码。一块代码被连接到了一个命令按钮上，用于测试控件向属性发送数据的速度：

```
Private Sub Commandl_Click()
Dim StartTime As Date
StantTime=Now
MousePointer=vbHourglass
Dim i As integer
for I=1 To 20000
  Faceoff. TestMe=i
Next i
MsgBOx "Operation took" +_Format&(Now-StartTime,"ss")+_ "seconds."
MousePointer=vbDefault
End Sub
```

另一块代码用于当表改变大小时及时改变控件的大小。它可用于观察控件的重画速度：

```
Private Sub Form_Resize()
Commandl. Top=ScaleHeight-(Command1. Height*1.5)
Command1. Left=ScaleWidth-Commad1. Width-300
FaceOff. Width-ScaleWidth-(300*2)
FaceOff. Ileight=Command1. Top-(300*1.5)
End Sub
```

接下来我开始速度测试。测试是很容易进行的。你只需将每个 VB 应用程序的名字拖到 LAUNCHEXE 上记下报告的加载时间；再点击 Test 按钮并改变表的大小。我是在8M 内存的386—40和8M 内存的486—100的机器上进行测试的。Microsoft 公司推荐的 Windows 95所用内存是12M，从我所接触的人都说如果有16M 内存会更好些，最好是有32M 内存。可是，我是一个穷作者，我可不象 Bill Gates(Microsoft 公司总裁)那么有钱。表14.6和14.7列出了测试结果。

表14.6　　三个 DLL 加载时间的比较

控件类型	386-40/8Mb	486-100/8Mb
VBX	14秒	10.5秒
16位 OCX	44秒	15秒
32位 OCX	13秒	

表14.7　**属性设置时间的比较**

控件类型	386-40/8Mb	486-100/8Mb
VBX	4	1
16位 OCX	11	3
32位 OCX	5	

我是在386机器上开发控件的，但当我试图将应用程序从386机器搬到486机器上(使用安装程序生成器)时，我又遇到了一个奇怪的问题：32位 OCX 没有自注册！这是因为安装程序生成器它自己本身有问题。但经过三周的研究，我仍没有发现问题的所在。所以32位 OCX 测试没有在486机器上进行。

该下结论了。我认为你最好只有当确实需要 OLE 控件的强大功能并愿意付出巨大努力研究未成熟的技术时，再去使用 OLE 控件。但如果你更愿意使用16位 VB4.0或 VB3.0(VB3.0应用程序加载速度要比 VB4.0应用程序快得多)，你根本就不需要 OLE 控件技术。

第十五章 创建一个简单的OCX

当你第一次编写OCX(OLE 控件)时，你会有坠入云里雾中那样的感觉。你知道得太少了，有太多的东西需要你学习，而太多的东西都隐藏在C++类和COM 说明中。虽然 Windows 已经够复杂的了，而 VBX 又包含了各种各样的知识，但至少这些知识是可以学会的。我们都希望不用真正了解底层框架的操作细节就可以创建OCX，可这是不现实的。我无法使OCX 变得简单。学习 OCX 虽然困难，但至少我可以陪你上路。有我形象地讲解，你不会孤独的。

我们要做的就是尽我们的所能用 Control Wizard 创建一个简单的OCX，再来看看 Control Wizard 所产生的代码。这部分代码是实现一个 OCX 最基本的代码，你可以用 Control Wizard 再加入更多的功能。

我在前面已提到过，有两种 OCX：16位的和32位的。16位 OCX 必须要使用 Visual C++ V1.52x 的 Control Wizard 来创建；更早版本的 Control Wizard 当然也可以用，但所生成的 OCX 不能在 Visual Basic4.0中使用。32位 OCX 必须要使用 VisualC++2.2或 VisualC++4.0 的 Control Wizard 来创键。由于 VC4.0是最新版本，所以我们使用它来生成32位 OCX。而16位的 OCX 是使用 VC1.52c 来生成的。

15.1 创建一个16位的 OCX

由于你需要 OLE 工具，所以你必须在安装完 Visual C++1.52c 之后，单独安装 Control Wizard。这次安装会把 Control Wizard 加入到 MSVC 的工具菜单中。无论你是否已经在 MSV-CIDE(集成开发环境)中打开了一个工程，启动 Control Wiazrd 都会新建一个工程。图15.1显示了它的主窗口的模样。

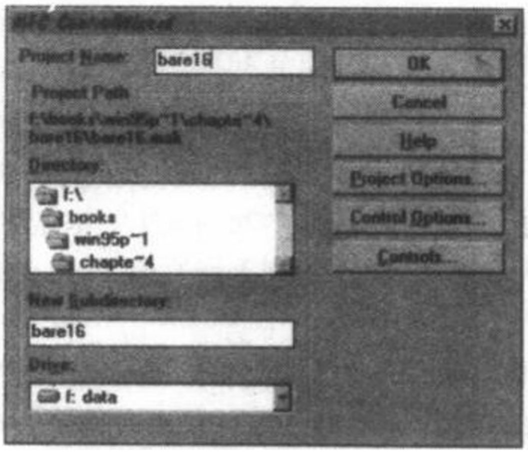

图15.1 用 Control Wizard 新建一个工程

工程名输入框的功能与 App Wizard 中的一样。并且与 App Wizard 一样，工程名被自动复制到新子目录域中。如果你愿意的话，你可以改变新子目录的名字，甚至可以把它设置为空而使用已有的目录。但最好不要用一个已存在的工程目录新建一个工程。Visual C++的各种 wizard(实际上就是代码生成器)一般在一个目录中只建一个工程。

还要注意的是，Visual C++1.5x 中的工具，包括 Control Wizard 并不能识别 Windows 95

中的长文件名。

点击 Project Options(工程选项)按钮会弹出一个 Project Options 对话框，见图15.2。

选择 Context Sensitive Help 复选框，告诉 Control Wizard 生成一组用于实现控件联机求助的文件。这些文件的格式是 RTF 格式。很少有人用这种方式来生成他们的帮助文件。但如果你愿意，这些文件是一个好的范本。

"External Makefile"(外部工程文件)是为那些仍习惯使用命令行的人准备的。选择"Source Comments"(源代码注释)会在源代码文件中包含一些有用的注释。你可能会选择这个选项，至少是在编写最初的几个 OCX 的时候。选择"License Validation"(许可确认)可以随工程生成一个．FIC 文件以及检验这个文件是否存在的代码。如果控件在运行时环境(Run-time environment)使用时，这个文件是否存在都无所谓；但如果控件在设计时环境(Design-time environment)使用时，这个文件必须存在。这可以允许你出售你编写的控件。如果使用这个控件的应用软件的终端用户不单独购买控件本身，就不能用它来进一步开发应用软件。

点击 Control Options 按钮会弹出 Control Options 对话框(见图15.3)。

图15.2 工程选项对话框

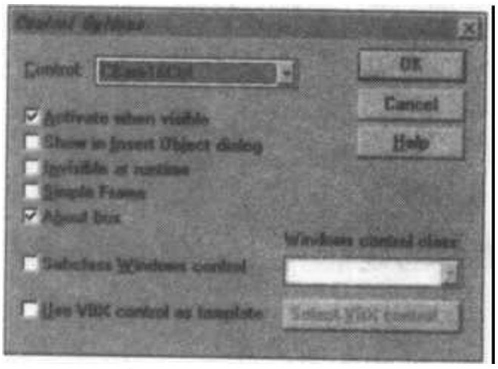
图15.3 控件选项对话框

这时在组合框中只有一个控件。如果你需要的话，你可以点击 Controls 按钮加入更多的控件。如果选择了"Activate when visible"(当可见时被激活)，最终的控件容器在这个控件处于可见状态时，要加载这个控件并使之处于激活状态。如果没选择这个选项，这个控件则更象一个非控件 OLE 服务器：如果用户或应用软件没有真正激活它，它将以默认方式显示。默认情况下，这个选项处于选择状态。

你一定见过 Insert Object(插入对象)对话框，任何 OLE 容器应用程序都有 Insert Object 这个菜单命令。这个对话框中包含有一个可以插入到容器中的 OLE 对象类型的列表。严格地说，OLE 控件就是一个 OLE 对象。但是，不管按钮或列表框做得有多漂亮，也很少有字处理文档的作者会想要在文档中加入一个铵钮或列表框对象。所以我们通常不会选择"Show in Inscrt Objcetdialog"。

我想你会猜得到"Invisiblc at runtimc"复选框的意思。但你要注意这只是一个建议，而不一定非要选择它。如果你选择了它，那么这个控件只有在设计模式时才会显示。

如果你使用过 Visual Basic，你就会知道边框控件(Frame control)是如何工作的：与标准 Windows 程序中只充当一种修饰的组框(Group box)不同，VB 边框控件实际上可以是其它控件的容器。如果一个边框控件被禁止了，那么这个边框所包含的全部控件都会被禁止；如果边框不可见了，边框中的控件也会不可见；如果边框被移动了，边框中的控件也会跟着移动等等。在 OLE 中是可以编写这种控件的，只需选择"Simple Frame"复选框就可以了。

About(关于)对话框是程序员检查控件版本和你要提供的有关控件其它方面信息的重要

途径。当然，终端用户从不会见到它。选择这个选项会自动生成About对话框文件并成为控件的一部分。注意：如果你用的是16位Control Wizard，而你又不喜欢它所生成的About对话框文件的风格，你是可以修改它的。

如果你的OCX是对标准Windows控件的改进(例如你想写一个定制的列表框或滚动范围允许超过32K的滚动条)，你可以选择“Subclass Windows control”选项并在“Windows control class”组合框中选择底层Windows控件。如果你想用你写或买的控件而不是Windows标准控件做为底层的子类控件，你可以在组合框中直接输入该控件名。(但你必须在控件中编写加载底层控件的DLL的代码。)但你要记住，因为你编写的是16位的OCX，所以你只能用16位的控件做为底层控件。这就漏掉了Windows 95为32位应用程序增加的那些新控件。

如果你想把现存的某个VBX转换成OCX，“Use VBX control as temple”按钮将会有很大帮助。当然接下来你要点击“Select VBX control”按钮来选择那个VBX。然后Control Wizard会装入这个VBX，读入它的MODEL结构、从VBX中复制控件名、属性名以及属性。当然，它并不能复制代码。这部分应由你来做，将VBX中实现那些属性和事件的代码复制过来不应该是件很复杂的差事吧！尤其是如果你的VBX是使用VBX Genie生成的，对操作属性的代码都编写在单独的函数中，那么复制这些代码应该是非常方便的。

类似于VBX，OCX可以包含多个控件。虽然与单控件VBX相比，多控件VBX的效率并没高多少；但与单控件OCX相比，多控件OCX的效率则更高得多；尤其是当应用软件同时使用多个控件时，更是如此。这是由于DLL加载时间造成的(前一章有详细介绍)。如果要描述OCX中的控件，你必须在Control Wizard对话框中点击Controls按钮，你就会看到一个控件对话框（见图15.4)。

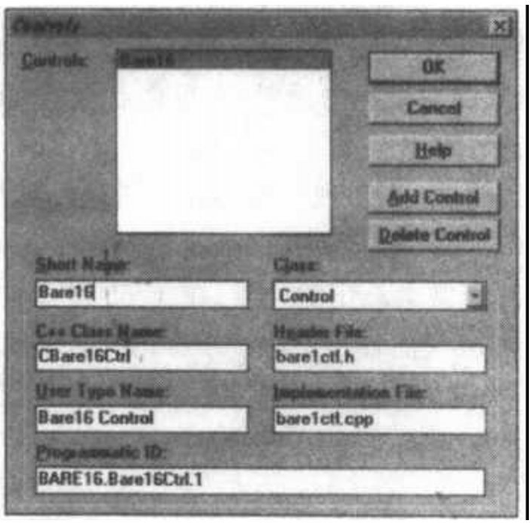

图15.4 使用控件对话框描述控件

当对话框出现时，它的列表框已经有了一个由工程名派生来的控件名。其它文本编辑框中的内容都是对这个控件的描述。要增加一个控件，只需点击Add control按钮。新的控件名与第一个控件名一样，只是在名字末尾添加了一个数字“1”。当然这不是个很好的名字，我们要给它重新起名。当你在列表框中点击某个表项时，下面的文本编辑框的内容也将随之改变。如果你想删除一个控件，只需选择它并点击Delete Control按钮即可。

“Short Name”的内容是OLE子系统内部所识别的控件名字。这个名字是系统注册(System Registry)时所使用的控件名字。而“User Type Name”的内容则不同，它是当用户向系统询问有哪些可用OLE控件时所呈现给用户的名字“C++Class Name”是定义该控件类的名

字，它的说明和实现分别放在你在“Header File”和“Implementation File”中输入的文件中。组合框 Class 中只包含有“Control”一个选项。

“Programmatic ID”中使用的是 OLE 自动化标识控件的标准格式：第一项是包含该控件的 OCX 的名字，接下来是它的另一个名字，最后一项看来是个常数；因为如果你增加一个控件，它的程序 ID 也是以“1”结尾的。

15.1.1 检查和改变控件设置

同 App Wizard 相似，Control Wizard 也会显示一个包含详细信息的对话框，在真正生成代码之前，它给你最后一次机会去重新考虑整个工程的设计。在这个例子中，我们全部使用了默认值。创建控件信息对话框显示在图15.5中。

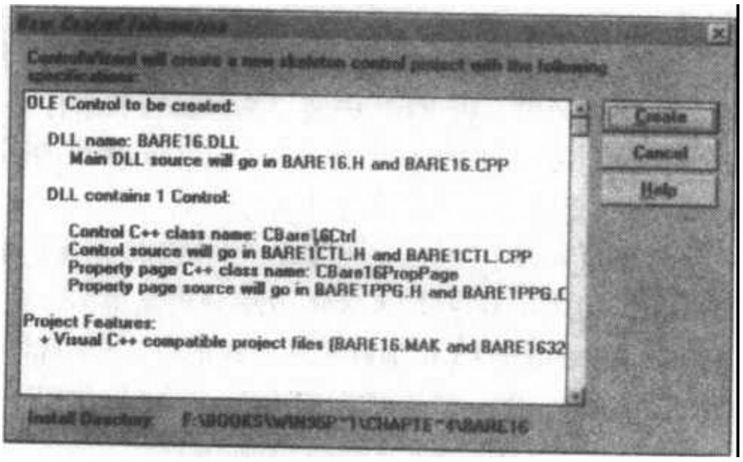

图15.5 使用创建控件信息对话框来检查控件的设置

你一定要仔细观察这里所呈现的信息。如果你现在点击 Cancel 按钮，你会回到 Control Wizard 对话框中去修改你遗忘的或错误的设置。一旦你点击了 OK 按钮，要想改变其中的任何设置，唯一的办法就是重来一遍。

当 Control Wizard 生成文件时，会出现一个小的状态对话框告诉你它在生成哪个文件。当这些文件都生成完毕了，Visual C＋＋会打开这个新工程，你就可以做后续工作了。

你的第一步工作也就是你通常会忘记的一步。在 Tools 菜单中除了 Control Wizard 外，还有一个 Make TypeLib 命令。在32位的环境中这一步骤已经自动执行了，但在16位的环境中这个命令必须显式执行。并且，当你向一个控件中增加属性、事件或方法后，必须在编译前再运行这个命令。这个命令所生成的文件叫做类型库(Type Library)。它用于将你的控件注册到 OLE 子系统中。

用 Control Wizard 生成代码后，在你想对 Control Wizard 所生成的框架做些改变之前，最好先编译一下整个工程。这使你有机会检验工程设置和目录位置等等，因为任何一点不合适都会引起编译或连接错误。

15.2 创建一个32位的 OCX

16位的 Control Wizard 在生成 OCX 的工程文件时，还生成了一个与32位 OCX 相匹配的 MAK 文件。你很容易发现它，因为它的名字末尾有数字“32”。比如，上一节所生成的 OCX 工程文件叫做 Bare16. MAK，但在同一目录下还有一个文件叫做 Bare 1632. MAK。但这个文件

是供 Visnal C++2.2使用的，而这个版本的 VC 已经过时了。如果你将它做为工作空间装入 Visual C++4.0，这个文件将会被转换并且在建造(build)工程之前一切都显得非常顺利。但在工程进行连接时会失败，因为找不到 OCS30D. LIB 这个文件。这个文件是 MFC3.0的一个组件，而 Visual C++1.52和4.0都不包含 MFC3.0。

将 BARE1632. MAK 文件中对“OCS30”引用改成对“OCS40”的引用仍没有效果。连接器还是提示同样的错误信息。我也没能在 IDE 中找到相对应的设置。

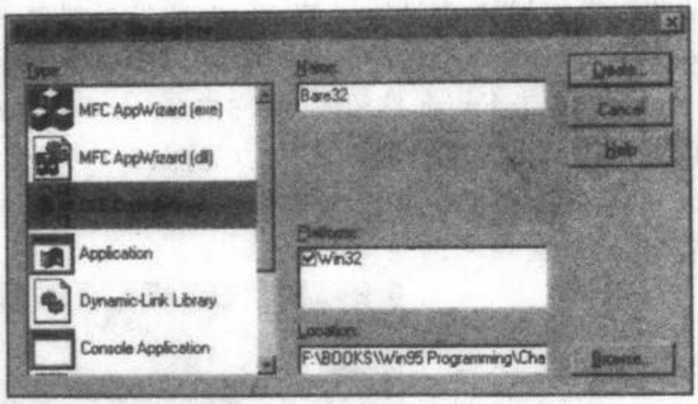

图15.6　使用 New Project Workspace 对话框创建工作空间

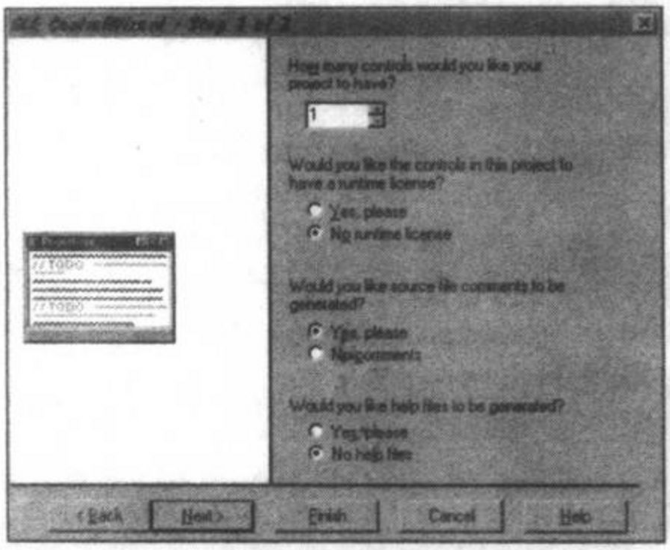

图15.7　使用 OLE ControlWizard 对话框

由此看来，建造32位的 OCX 必须要从头开始。幸好，这不是件艰苦的工作。毕竟，32位的 Control Wizard 为我们做了大部分的工作。并且我在后面会给你提供了一些建议以便更容易同时维护两种版本的 OCX 代码。

在 Visual C++4.0中 Control Wizard 已不再是单独的工具。要创建一个 OCX 工程，你可以使用 File 菜单，选择 New…命令。OCX 工程的创建方法和其它类型的工程创建方法是一样的。运行 New…命令会出现一个创建对话框要你选择工程的类型，你只需选择其中的一种，点击 OK 即可。

创建工程工作空间(New Project Workspace)对话框是你选择工程类型的地方。“OLE ControlWizard”是其中的一个选项(见图15.6)。

在这个例子中，Create 按钮并不马上创建工程，而是弹出第一个 Control Wizard 面板。该版本的 Control Wizard 使用多个页面将信息呈现给你，让你设置其中的选项(见图15.7)。

当然这里所呈现的选项与16位版本中的一样。你要回答的第一个问题：你打算在这个 OCX 中放入多少控件?你可以直接输入控件数(从1到99个)。

在下一页中包含两个重要的内容(见图15.8)。第一个是你有机会更改控件、类和文件的默认名。点击 Edit Name 按钮会显示一个对话框(见图15.9)。这个对话框中不但包含有16位 Control Wizard 所提供的内容，还包含有16位 Control Wizard 没提供的有关属性页(Property Page)的描述。第二个是呈现

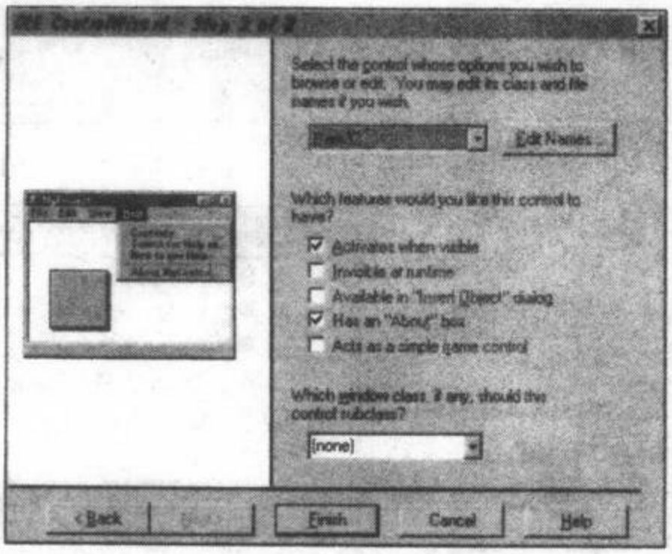

图15.8　OLE Control Wizard 对话框的第二页

了可以做为OLE控件底层Windows控件类列表。除了原有的Windows类外，这个列表中还包含有Windows 95所提供的全部新类。表15.1显示了全部列表。

当你点击Finish按钮时，和16位Control Wizard一样，你还有最后一次机会检查那些控件设置以确保它们都是你所想要的(见图15.10)。和以前一样，如果你对一些设置不满意，你可以点击Cancel按钮返回，再继续修改。如果你点击OK按钮，你的工程就会被生成(见图15.11)。

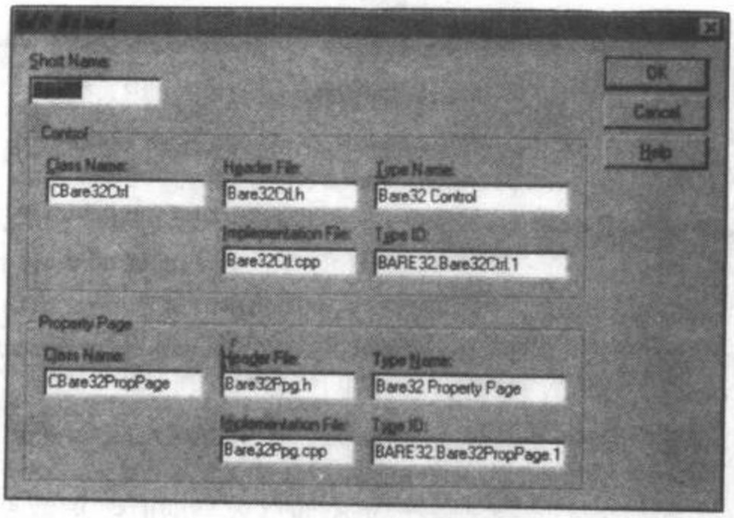

图15.9　Edit Names对话框

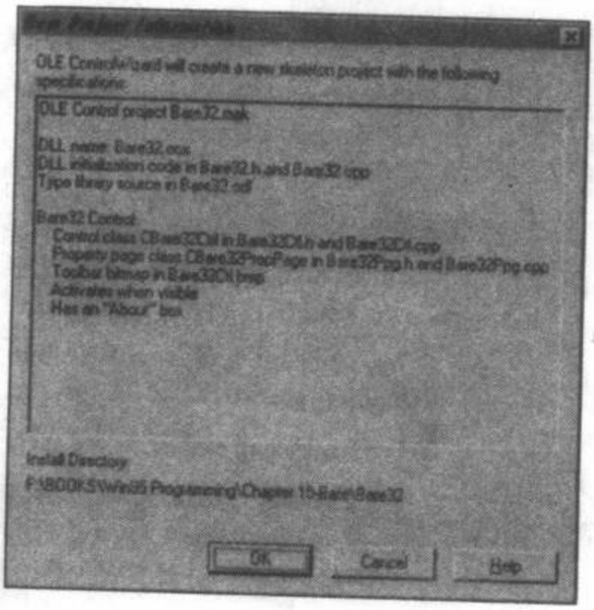

图15.10　检查创建信息对话框

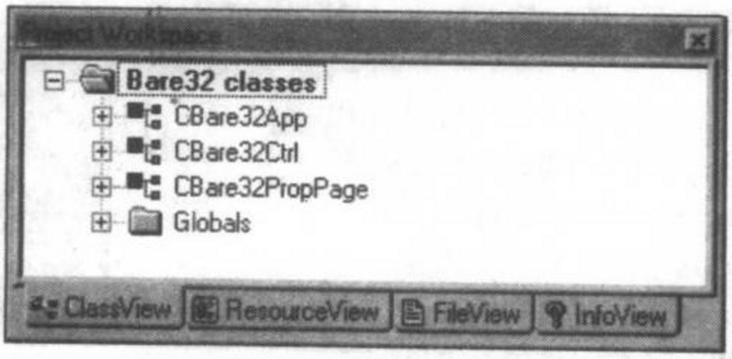

图15.11　生成最终的工程

这时，你就可以建造(用 build 命令)你的工程了。首次建造会很费时间，这要看你的计算机中有多少内存。甚至会需要你关闭其它应用程序。但经过首次建造后，只要你保存好各种中间文件，后面因少量修改代码而引起的重建(用 rebuild 命令)会很省时。

窗口类	描述
BUTTON	原来的 Windows 按钮类。这个类仍包含有各种传统的风格，如：radiobutton (圆按钮)，check-box (复选框)，group box frame (组框) 等。
COMBOBOX	包含三种风格的传统组合框：simple、drop down (下拉型) 以及 drop down list (下拉列表型)
LISTBOX	包含多种风格的列表框：single、multiple and extended selection (多选和扩展选择型)，multi-column (多列型) 等等。
msctls_hotkey32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。它是一个能让用户定义热键的窗口。
msctls_progressbar32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。在应用程序中它通过画彩色矩形条来标志长时间操作的执行进度。
msctls_statusbar32	它只能在32位的 OCX、Windows 95以及 WindowsNT4.0中使用。它是一个水平窗口，在应用程序中可以用来显示各种状态信息。这个控件与 MFC 中的 CStatusBar 类很相似，所以在16位的 OCX 应该能够模拟它。
msctls_trackbar32	它也叫滑动条 (slider)。它只能在32位的 OCX、Windows95 以及 Windows NT4.0中使用。它是一个包含有一个滑动条和滑动柄的窗口：当滑动柄的位置变化的时候，该控件会向其父窗口发送相应的通知消息。
msctls_updown32	它只能在32位的 OCX、Windows 95以及 WindowsNT4.0中使用。它是一对箭头按钮。用户可以点击按钮来增加或减少某个数值，如滚动位置或显示的数字等。
SCROLLBAR	指传统的水平和垂直滚动条。
STATIC	传统的静态控件，它可以显示文本或图标。
msctls_SysAnimate32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。它是一个能播放 AVI (Audio Video Interleaved) 文件的窗口。
msctls_Header32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。它是一种出现在一列文本上方的可变大小的按钮。
msctls_ListVlew32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。现在还没有关于这种控件的详细文档资料。
msctls_TabControl32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。它是一种类似于笔记本标签的按钮。应用程序使用它可以将同一区域定义成多个页面。
msctls_TreeView32	它只能在32位的 OCX、Windows 95以及 Windows NT4.0中使用。它是一个能显示表项层次列表的窗口，如文档的标题、目录的索引项或磁盘中的文件列表等。每个表项都有一个标志和可选位图以及相关子项与它相联系。

15.3 检查代码

由于我们生成了两个 OCX，一个是16位的，另一个是32的。所以我想逐个检查一下它们的代码会对我们很有帮助。

与它所包含的控件不同，一个 OCX 是由单个模块来表征的。在这个例子中，这个模块的头文件分别是 BARE16.H 和 BARE32.H。除了名字不同外，内容是一样的：

```
#bare16. h main header file for BARE16. DLL
```

```
#fldefined(_AFXCTL_H_)
    #error include 'afxctl. h' before including this file
#endif
#include "resource. h" #main symbols
#CBare16App: See bare16. cpp for implementation.
class CBare16App: public COleControlModule
 {
public:
  BOOL Initinstance();
  int Exitinstance();
};
extern const GUID CDECL_tlid;
extern const WORD_wVerMajor;
extern const WORD_wVerMinor;
```

你可以在 Control Wizard 生成这个类以前改变类的名字。但很少有人这样做，因为人们很少真正关心这段代码。

COleControlModule 类以前是 MFC2.5的一种扩展，现在已成为 MFC4.0的一个组成部分。顾名思义，该类封装了 OLE 控件。在一个实际的控件中只有 InitInstance()和 ExitInstance()方法必须被重载。Control Wizard 在 BARE16. CPP 和 BARE32. CPP 中就是这样做的：

```
#bare16. cpp: Implementation of CBare16App and DLL registration.
#include "stdafx. h"
#include "bare16. h"
#ifdef_DEBUG
#undef THIS_FILE
static char BASED_CODE TIHS_FILE [] =_FILE_;
#endif
CBare16App NEAR theApp;
const GUID CDECL BASED_CODE_tlid=
 {0xbf3c5063, 0x5433, 0x11cf, {0xb4, 0x27, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0}};
const WORD_wVerMajor=1;
const WORD_wVerMinor=0;
#Bare32. cpp: Implementation of CBare. 32App and DLL registration.
#include "stdafx. h"
#include "Bare32. h"
#ifdef_DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char TIHS_FILE [] =_FILE_;
#endif
CBare32App NEAR theApp;
```

```
const GUID CDECL BASED_CODE_tlid=
  {0x8af31f00, 0x54b0, ox11cf, {0xb4, 0x27, 0x44, 0x45, 0x53, 0x54, 0, 0}};
const WORD_wVerMajor=1;
const WORD_wVerMinor=0;
```

在这两个文件中都生成了一个 COleControlModule 派生类的实例。文件中定义了一个用模块全局 ID 初始化的常量，并且常量中的数值随模块的不同而不同。(如果你重新生成这个工程，那么它的 GUID 值还会变化。)主版本号和次版本号也被定义成常量，并且可以作用于整个工程，因为它们在头文件中被说明成了外部变量(extern)。要注意，这些值不一定会与 VERSIONINFO 资源中版本域的值保持一致。所以如果你改变了资源，不要忘记还应改变这些常量值。

除了名字外，两个 OCX 中的 InitInstance()和 ExitInstance()是一样：

```
BOOL CBare16App:: InitInstance()
 {
  BOOL bInit=COleControlModule:: InitInstance();
  if(bInit)
   {
    # TODO: Add your own module initialization code here.
   }
   return bInit;
    {
int CBare16App:: ExitInstance()
 {
  # TODO: Add your own module termination code here.
  return COleControlModule:: ExitInstance();
 }
```

如果你想在 OCX 被加载或下载时做一些特别的操作(如显示版权信息等)，你就需要改变这两个函数。但一般的情况下，你完全没必要理睬它们。

OCX 是自注册的，并且注册函数不属于任何类。但它被定义在模块的实现文件中：

```
STDAPP DllRegisterServer(void)
 {
AFX_MANAGE_STATE(_afxModuleAddrThis)
if(!AfxOleRegisterTypeLib(AfxGetinstanceHandle(),_tlid))
  return ResultFromScode(SELFREG_E_TYPELIB);
if(!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
  return ResultFromScode(SELFREG_E_CLASS);
return NOERROR;
 }
```

“Scode(范围)”是一种结构。它不是简单地返回“失败”或“成功”，而是返回从“完全失败”、“有些失败”、“有些成功”到“完全成功”范围中的某个值。但实际上这个复杂的数值包含有一些冗余。所以 ResultFromScode()宏将它简化成简单的 TRUE 或 FALSE。

有时需要用下面的函数注消 OCX：

```
STDAPI DllUnregisterServer(void)
 {
AFX_MANAGE_STATE(_afxModuleAddrThis);
if(!AfxOleUnregisterTypeLib(_tlid))
  return ResultFromScode(SELFREG_E_TYPELIB);
if(!COleObjectFactoryEx::UpdateRegistryAll(FALSE))
  return ResuitFromScode(SELFREG_E_CLASS);
return NOERROR;
}
```

控件本身说明在一对文件中。这对文件的名字当然是从 OCX 的名字派生出来的。在实际的开发中，你可能会改变它们。但我在这里没有对它们做任何改动，以便我们来看看它们到底是什么。因为16位的 Control Wizard 对文件的命名严格遵守8.3规则(即名字最多是8个字母；而后缀最多有3个字母)，所以 BARE16CTL. H 变成了 BARE1CTL. H。而32位的 Control Wizard 允许长文件名，所以 BARE32CTL. H 名字没有变。下面是文件的内容：

```
//bareictl. h: Declaration of the CBare16Ctrl OLE control class.
//CBare16Ctrl: See bare1ctl. cpp for implementation.
class CBare16Ctrl: public COleControl
 {
DECLARE DYNCREATE(CBare16Ctrl)
//Constructor
public:
  CBare16Ctrl();
//Overrides
//Drawing function
virtual void OnDraw(CDC*pdc,const CRect &rcBounds,const CRect & rcInvalid);
//Persistence
virtual void DoPropExchange (CPropExchange * pPx);
//Reset control state
virtual void OnResetState();
//Implementation
protected:
  ~CBare16Ctrl();
DECLARE_OLECREATE_EX(CBare16Ctrl)  //Class factory and guid
DECLARE OLETYPELIB(CBare16Ctrl)  //GetTypeInfo
DECLARE_PROPPAGEIDS(CBare16Ctrl)  //Property page lds
DECLARE_OLECTLTYPE(CBare16Ctrl)  //Type name and misc status
//Message maps
// { {AFX_MSG(CBare16Ctrl)
      //NOTE-Class Wizard will add and remove member functions here.
```

```
        //DONOTEDIT what you see in these blocks of generated code!
    //}} AFX MSG
    DECLARE_MESSAGE_MAP()
    //Dispatch maps
    // { {AFX_DISPATCH(CBare16Ctrl)
        //NOTE-ClassWizard will add and remove member functions here.
        //DONOTEDTT what you see in these blocks of generated code!
    //}} AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    afx_msg void AboutBox();
    #Event maps
    // { {AFX_EVENT(CBare16Ctrl)
        //NOTE-ClassWizard will add and remove member functions here.
        //DONOT EDIT what you see in these blocks of generated code!
    //}} AFX_EVENT
    DECLARE_EVENT_MAP()
      //Dispatch and event Ids
    public:
      enum {
      // { {AFX_DISP_ID(CBare16Ctrl)
          #NOTE: ClassWizard will add and remove enumeration elements here.
          #DO NOT EDIT what you see in these blocks of generated code!
      //}} AFX_DISP_ID
      };
    };
```

Control Wizard 非常喜欢在所生成的代码中使用大量的宏来隐藏具体的操作。我猜想这可能有助于 Control Wizard 避免产生代码拼写错。同使用 App Wizard 一样，非常关键的一点是你在修改代码时一定要遵循注释部分所给你的建议。你可以修改 Class Wizard 为你生成的代码。凭经验我认为改变这部分代码的格式并不会有什么害处。并且你当然可以在类中增加你所需要的方法原型。但我建议你最好不要随便删除 Control Wizard 为你生成的方法或属性。

控件的实现分别放在文件 BAREICTL. CPP 和 BARE32CTL. CPP 中。这两个文件都以支持对象动态生成机制的宏开始：

```
//bareictl. cpp : Implementation of the CBate16Ctrl OLE control class.
//include "stdafxh"
//include "bare16. h"
//include "bare1ctl. h"
//include "barelppg. h"
//ifdef_DEBUG
//undefTHIS_FILE
static char BASED_CODE TIHS_FILE [] =_FILE_;
```

```
//endif
IMPLEMENT_DYNCREATE(CBare16Ctrl,COleControl)
```

消息映射(Message map)是指将对 Windows 消息或菜单命令的处理映射到某个对象的成员函数上去。当然在 OLE 控件中并没有来自菜单的消息，并且控件本身只提供对来自窗口的消息处理。所以这里使用消息映射机制看起来好象没必要，但它却是 MFC 处理消息的方式并且 Class Wizard 好象非常喜欢它：

```
//Message map
BEGIN_MESSAGE_MAP(CBare16Ctrl,COleControl)
  //{{AFX_MSG_MAP(CBare16Ctrl)
  //NOTE-ClassWizard will add and remove message map entries
  //DO NOT EDIT what you see in these blocks of generated code!
  //}} AFX_MSG_MAP
  ON_OLEVERB(AFX_IDS_VERB_PROPERTIES,OnPropeities)
END_MESSAGE_MAP()
```

发送映射(Dispatch map)的概念与消息映射是相似的，所不同的是它将 OLE 消息通过接口发送给 MFC。换句话说，当你的控件容器设置属性或调用方法时，发送映射会确保执行正确的代码：

```
//Dispatch map
BEGIN_DISPATCH_MAP(CBare16Ctrl,COleControl)
  //{{AFX_DISPATCH_MAP(CBare16Ctrl)
  //NOTE-ClassWizard will add and remove dispatch map entries
  //DO NOT EDIT what you see in these blocks of generated code!
  //}} AFX_DISPATCH_MAP
DISP_FUNCTION_ID(CBare16Ctrl,"AboutBox",DISPID_ABOUTBOX,AboutBox,
VT_EMPTY,VTS_NONE)
END_DISPATCH_MAP()
```

事件同样也是 OLE 通过接口进行管理的。但因为事件都是由控件激发的，而不是由控件的操作激发的，所以单独存在着一个事件映射(Event map)：

```
#Event map
BEGIN_EVENT_MAP(CBare16Ctrl,COleControl)
  //{{AFX_EVENT_MAP(CBare16Ctrl)
  //NOTE-ClassWizard will add and remove event map entries
  //DO NOT EDIT what you see in these blocks of generated code!
  //}} AFX_EVENT_MAP
END_EVENT_MAP()
```

15.4 支持属性页

现在你应该已经熟悉 Visual Basic 中的工具箱了。同时你也知道 Visual C++4.0的 App Studio 在设计对话框或其它表时也有一个工具箱。一个 OCX 在被加入到用户控件列表后会出

现的工具箱中。在 Visual Basic 中控件的属性被列在属性框中：而 Visual C++则使用属性页(property page)。属性页是 Windows 95程序可以创建的一种新型资源。

属性页是对话框中的一组标签。每个标签都有一个属性所在组的名字，如“Font(字体)”、“Color(颜色)”或其它什么的。在设计阶段，OLE 控件是通过属性页进行修改的，所以有一个结构来定　之们：

```
//Property pages
//TODO: add more property pages as needed.  Remember to increase the count!
BEGIN_PROPPAGEIDS(CBare16Ctrl,1)
  PROPPAGEID(CBare16PropPage::guid)
END_PROPPAGEIDS(CBare16Ctrl)
```

你提供的页的默认名为“General”：它是在对话框中用户所希望见到的第一个页上。有一对库存属性页你可以使用，其它属性页则需要你去设计(我们在下一章将会做这些工作)。

15.5 支持注册数据库

下一段代码中标识符(即控件各方面的 GUID)包含着有关注册数据库的信息：

```
//Initialize class factory and guid
IMPLEMENT_OLECREATE_EX(CBare16Ctrl,"Bare16. Bare16Ctrl. 1",
  0xbf3c5060,ox5433,0x11cf,0xb4,0x27,0x44,0x45,0x53,0x54,0x0,0x0)
//Type library ID and version
IMPLEMENT_OLETYPELIB(CBare16Ctrl,tlid,_wVerMajor,_wVerMinor)
//Interface Ids
const HDBASED_CODEHD_DBare16=
 {oxbf3c5061, 0x5433, 0x11cf, {0xb4, 0x27, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0}};
const HD BASED_CODEHD_DBare16Events=
 {oxbf3c5062, 0x5433, 0x11cf, {0xb4, 0x27, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0}};
```

注意一下对_wVerMajor 和_wVerMinor 引用。这些常数值独立于 VERSIONINFO 资源中相应域的数值。资源中这些数值可用于在用户机器中重新安排使用 OCX 资源。

在生成控件时你可以规定多种选项。下面是典型的设置：

```
//Control type information
static const DWORD BASED_CODE_dwBare16OleMise=
OLEMISC_ACTIVATEWHENVISIBLE|
OLEMISC_SETCLIENTSITEFIRST|
OLEMISC_INSIDEOUT|
OLEMISC_CANTLINKINSIDE|
OLEMISC_RECOMPOSEONRESIZE;
IMPLEMENT_OLECTLTYPE(CBare16CtrlIDS_BARE16_dwBare16OleMisc)
```

头文件中有一个宏用于分别在 CBare16Ctrl 和 CBare32Ctrl 类中生成嵌入类(embedded class)。它是一个类厂(class factory)，OLE 实际上需要用它来生成控件的实例。大多数类厂都是通过宏实现的，但必须显式提供一个方法：

```
//CBare16Ctrl::CBare16CtriFactory::UpdateRegistry
//Adds or removes system registry entries for CBare16Ctrl
BOOL CBare16Ctrl::CBare16CtrlFactory::UpdateRegistry (BOOL bRegister)
 {
if(bRegister)
  return AfxOleRegisterControlClass(
        AfxGetInstancellandie(),
        m_clsid,
        m_lpszProgID,
        IDS_BARE16,
        IDB_BARE16,
        FALSE,
        _dwBare16OleMisc,
        _tlid,
        _wVerMajor,
        _wVerMinor);
else
  return AfxOleUnregisterClass(m_clsid,m_lpszProgID);
}
```

记住，这里我们是在注册控件，前面我们见过的注册代码是注册 OCX。

现在我们该接触到我们可以真正修改的代码了。首先是控件的构造函数和析构函数：

```
//CBare16Ctrl::CBare16Ctrl-Constructor
CBare16Ctrl::CBare16Ctrl()
 {
InitializeIIDs(&IID_DBare16,&IID_DBare16Events);
//TODO: Initialize your control' s instance data her.
}
//CBare16Ctrl::~CBare16Ctrl-Destructor
CBare16Ctrl::~CBare16Ctrl ()
 {
//TODO: Clean up your control's instance data here.
}
```

通常，在构造函数中你分配控件所需要的特殊资源；而在析构函数中再释放它们。

OnDraw()函数的功能与 MFC 中的 CWnd 派生类中的同名函数是一样的：显示控件。你应在这里完成全部的作图操作。无论什么时候，只要需要作图就可以调用 OnDraw()函数。即使不是在屏幕上或控件不处于激活状态下也可以调用这个函数。Control Wizard 所生成的 OnDraw()实现代码只是简单地画一个矩形并在其中画一个椭圆。你当然需要在实际开发控件时代替这部分代码，但在这里，这些代码足够了：

```
//CBare16Ctrl::OnDraw-Crawing function
void CBare16Ctrl::OnDraw(CDC * pdc,const CRect & rcBounds,const CRect& rcInvalid)
```

```
{
//TODO: Replace the following code with your own drawing code.
 pdc->FillRect(rcBounds,CBrush::FromHandle(HBRUSH)GetStockObject(WHITE-
BRUSH)));
pdc->Ellipse(rcBounds);
}
```

就象 MFC 为对话框中的控件与对象变量之间的数据自动传输提供函数一样，Control-Wizard 也为属性页的数据传输提供了相应的函数：

```
//CBare16Ctrl:: DoPropExchange-Persistence support
void CBare16Ctrl:: DoPropExchange(CPropExchange * pPX)
{
ExchangeVersion(pPX,MAKELONG(_wVerMinor,_wVerMajor));
COleControl:: DoPropExchange(pPX);
//TODO: CallPX_functions for each persistent custom property.
}
```

当你向你的控件中增加属性时，你可能会改变属性页模板并且加入一些代码以实现那些数值与控件之间的数据传输。在下一章我们在处理属性页时会接触到这些操作。

在 OLE 世界中有些事情的发生我们是无需知道其中的原因的。比如，有时一个运行控件必须能够重置它的初值。方法 OnResetState()可以完成这个工作：

```
//CBare16Ctrl:: OnRcsctStatc-Rcscl control to default statc
void CBare16Ctrl:: OnResetState()
{
COleControl:: OnResetState(); #Resets defaults found in DoPropExchange
#TODO: Reset any other control state here.
}
```

当你向控件中加入属性时，你可能会在这个方法中而不是构造函数中给属性赋初值。构造函数只会执行一次，而这个方法则可以运行多次。

如果你想显示 About 对话框，则运行下面的方法：

```
//CBare16Ctrl:: AboutBox-Display an "About" box to the user
void CBare16Ctrl:: About Box()
{
CDialog dlgAbout (IDD_ABOUTBOX_BARE16);
dlgAbout. DoModal();
}
```

这个函数是文件中最后一个函数，其它部分都是注释：

```
//CBare16Ctrl message handlers
```

如果你需要的话，Control Wizard 会在这里加入一些消息处理器。

你可能会注意到了，除了类名和 GUID 外，16位控件和32位控件的其它方面都是一样的。但由于这两个版本的控件 GUID 必须是不同的(否则会引起注册冲突)，所以你不可能用同一

个文件来同时代表两个版本的控件。你可以这样做：将相同的代码放在单独的文件中，并在工程中适当的地方包含它。比如

```
void CBarel6Ctrl:: OnDraw (CDC * pdc, const CRect & rcBounds, const CRect & rcInvalide)
{
#include "ondraw. inc"
}
```

还有另外一个实现属性页对话框的文件，但我打算在下一章处理属性页时来介绍它。然而还有一对非常重要的文件：STDAFX. H 和 STDAFX. CPP。这两个文件中并不包含有真正的代码。STDAFX. CPP 只包含一个 #include "stdafx. h" 语句。而 STDAFX. H 文件中包含了全部的标准 AFX 头文件。AFX 来自于 MFC 最初名字："Application Frameworks"。Microsoft 之所以没有使用这个名字是因为 Borland 公司抢先使用这个名字发行了一种软件产品。STDAFX. H 文件主要用于生成预编译头。你在建造工程的时候可以看到：STDAFX. CPP 经常首先被单独编译，而且也最费时。但这是值得的，因为后续的实现代码的编译都可以直接重新使用这个预编译头。

虽然 Microsoft 公司费尽力气所建造的 OLE 子系统的确令人望而生畏，但我却希望你能对以上生成代码的工作感到很容易。如果你熟悉 Control Wizard 生成的 MFC 代码，那么修改那些代码就会变得尤如散步在夏天的林荫小道上那样惬意而不会象走在黑暗密林中那样可怕了。

第十六章　使用 OCX 库存属性

16.1　设计一个更好的列表框

虽然 Visual Basic 隐藏了标准 Windows 列表框的许多很棒的功能(如按 TAB 键改变输入焦点和拥有者自画等),但改进 Visual Basic 的列表框也并非难事。既然我们想在学习如何编写 OLE 控件的过程中练练手，为什么不先从改进 VB 列表框开始呢?

在这一章，我们要重点讲解 OCX 库存属性的实现。我们将把那些库存属性加入到同一 OCX 中的三个不同但相关的控件中。这三个控件将分别成为三种特殊的列表框。第一个 TabList 是普通的文本列表框，但它支持用 Tab 键在列信息间改变输入焦点；第二个 ImageList 是一个图片列表框，它显示的是图片而不是文本；第三个 DrawList 将支持拥用者自画功能。

这听起来有意思吗?好!我们开始吧。

运行16位的 Control Wizard，并且将工程命名为 “Listbxs”。New Subdirectory 域默认的值是工程名(见图16.1)。

Control Options 和 Controls 这两个按钮的位置放得很奇怪，因为你应首先点击底下的 Control…按钮。点击它会弹出一个对话框，你在这个对话框中定义 OCX 所包含的控件(见图16.2)。

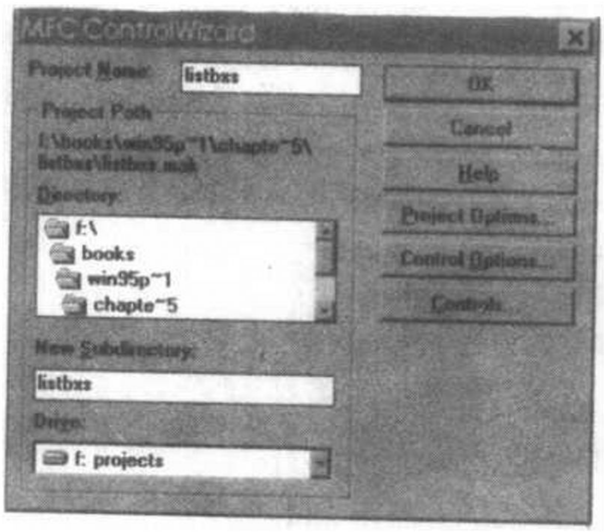

图16.1　创建一个新工程

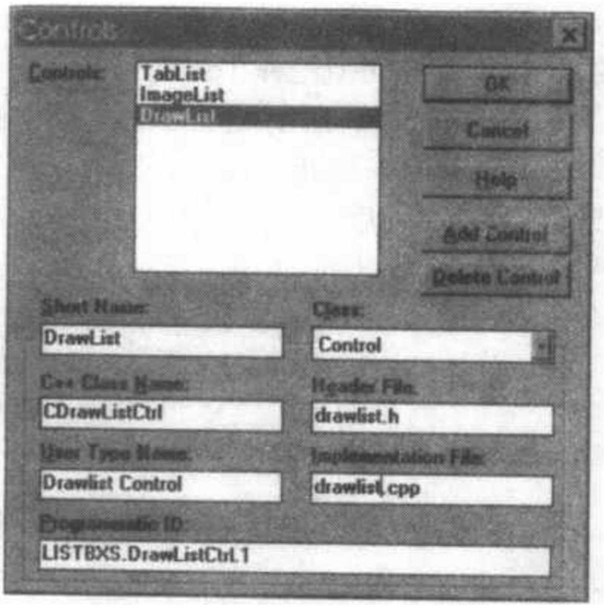

图16.2　对 OCX 选择控件

定义完那三个控件后，你要点击 Control Options…按钮，它弹出一个对话框(见图16.3)。这个对话框中一个重要的事情是这三个控件都要以 Windows 列表框控件作为底层控件(即子类 subclass)。

完成了以上的工作后，你就可以生成工程了。在生成工程后，别忘了要先运行 Make Typelib 工具，然后再进行工程的编译。

但在编译工程前，你可能想改变 Control Wizard 为控件工具箱所提供的位图。默认情况

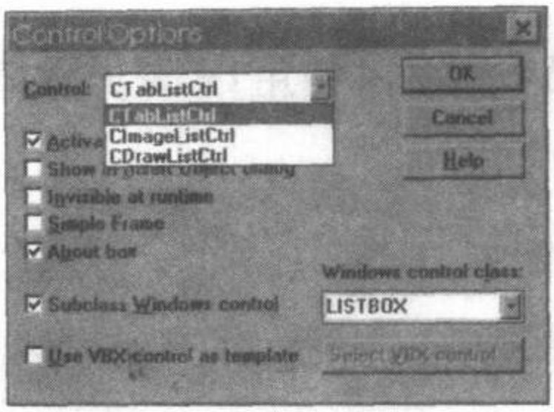

图16.3 Control Options 对话框

下，Control Wizard 为 OCX 中的每一个控件都生成一个带有“OCX”字符串灰色的方块图。这使它们很难相互区分，所以我自己生成了一组位图，见图16.4、图16.5和图16.6。

图16.4 TabList 按钮

图16.5 ImageList 按钮

图16.6 DrawList 按钮

你会注意到 OCX 按钮位图与 VBX 按钮位图不同，它并不需要按钮边界，OCX 容器的工具箱会提供按钮边界。

16.2 试运行 TabList

一旦这个工程通过编译，你就可以试验这个新的 OCX 控件了。运行 VB4-16(Visual Basic 4.0的16位版本)并且选择 Tools Custom Controls 命令。将会出现注册控件的列表，但你的新 OCX 并不会列在那里，因为它还没注册呢。由于 OCX 是自注册的，所以你所要做的是加载它一次。点击 Browse 按钮并找到你的新 OCX(Visual C++将生成 LISTBXS.DLL 文件，我们需要手工将这文件后缀改为 OCX)。选择这个 OCX 后，它就会出现在 Custom Control(用户控件)对话框中(见图16.7)。并且一旦注册，控件就将一直出现在这个对话框中，直到它被注销

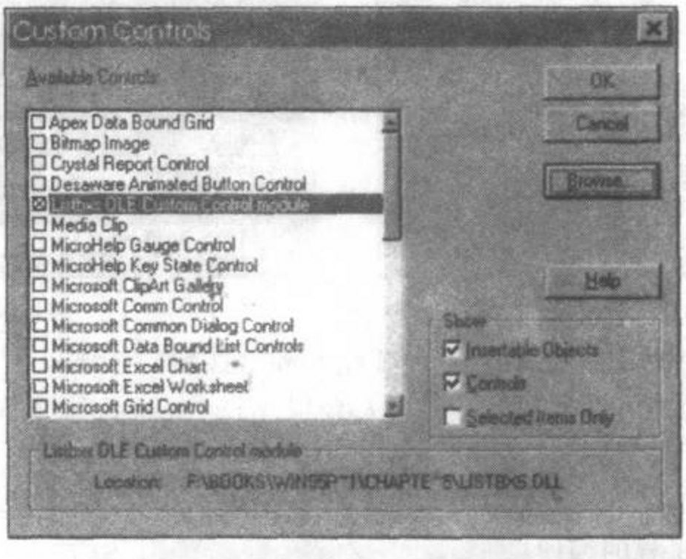

图16.7 Visual Basic 用户控件对话框

为止。

顺便提一下，如果你碰巧将一个16位控件加载到32位 Visual Basic 中，你会得到一个简单的表明不能加载控件的出错信息。如果你安装了两个版本(16位的和32位的)的 VB4，你一定要注意，VBP 文件只与你最后一次安装的 VB4相关联。你可以使用 Explorer 的 Options(选项)中的 File Types(文件类型)对话框来改变这个文件后缀。比如，我将 open 动词指向 VB4-16，而增加了一个 open 32-bits 动词来应付其它的情况。

16.3 了解隐藏的控件

一旦 LISTBXS.OCX 被加入到你的工具列表中，那三个新位图会立刻出现在你的工具箱中。通过它们你可以将控件加入到表(form)中。但当你这么操作时，你首先可能觉得什么都没发生。因为虽然你可以画出这个新控件，但在设计时环境中你只能看到一个改变控件大小的边框。而且如果不选择这个控件，那么连这个边框也消失了。但控件确实就在那里。并且如果你运行这个工程，你就会看到这个控件将显示成一个空的列表框，并且这个列表框没有边框。

我们已经生成了三个控件并编译了它们，但并没有定义任何属性。现在，我们可以使用一些默认的属性。当你选择控件时，你可以在 VB 属性框中看到这些属性。但它们都是由容器来管理的“环境”属性，如位置和大小属性等。BorderStyle(边框风格)并不在其中，因为控件的边框必须由它自身来管理。

要向控件中加入 BorderStyle 属性以及其它一些库存属性，你必须运行 Visual C++Class Wizard。选择 OLE Automation 页面，并确保需要增加属性的控件列在 Class Name 组合框中。(我们将分别向这三个控件加入属性，但你必须对每一个控件都进行一遍相同的操作过程。)点击 Add Property…按钮，它会弹出一个对话框(见图16.8)。前一章已提到过，External Name 组合框中预装入了一个可用的库存属性列表，BorderStyle 就是其中的一个。这个 OCX 中三个控件应加入的属性显示在图16.9。

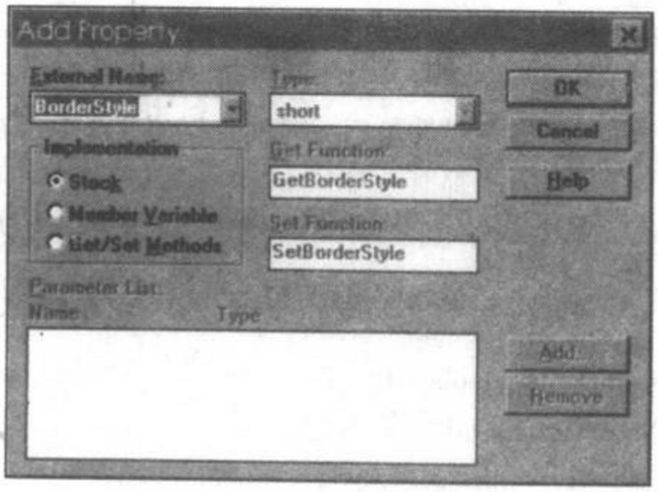

图16.8 增加属性对话框

如果你现在就编译，这些控件属性是不会出现在 VB 属性框中的。为什么会这样?很简单，因为它们还没注册!你必须再一次地运行 Tool…MakeTypelib 命令，而后编译工程。每当你向你的 OLE 控件中加入新的属性、事件或方法时，你都必须这么做一次。一旦类型库(typelib)被更新了，那些新属性就会在下一次加载 OCX 时被加入到注册数据库中，BorderStyle 属性就会

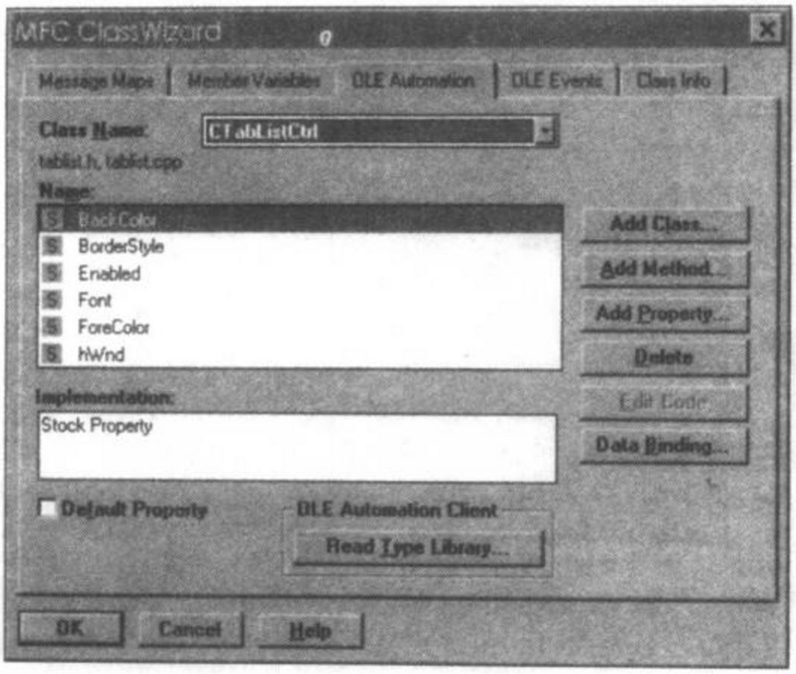

图16.9　增强列表框控件功能的库存属性

出现在属性框中。但它的默认值是0，即没有边框。我们如何改变它呢？

16.4　设置默认的属性值

库存属性由我们不能直接操作的基类来管理。那么，你如何来改变库存属性的默认值呢？

严格地说，你做不到！但对于某个特定的控件来说，你可以改变它们的默认值。你只需在控件第一次生成时，将属性设置为需要的数值。不要将控件本身的生成与代表控件的C++对象的构造函数相混淆。有一个方法叫OnResetState()。当控件无论因为什么原因而需要重置时，这个方法都会被调用。其中当控件第一次生成时，就会调用这个方法。那么，我来问你，如何设置控件属性值呢？你该知道了吧！

你再看看图16.8，你会注意与库存属性BorderStyle相对应地有一对Get/Set函数。函数名呈禁止状态，不能被改变。这是因为属性是库存属性，而那些函数是在基类中实现的。它们负责管理库存属性。每个库存属性分别都有一对Get/Set函数。所以要将BorderStyle属性默认值设为1(固定单线边框)，你必须在这三个控件中的OnResetState()方法中加入适当的代码：

```
void CTabListCtrl::OnResetState()
 {
//Invocation of base method supplied by
Control Wizard COleControl::OnResetState();
//Specific default property values setby you
SetBorderStyle(1);
}
```

编译后，如果你向VB表中加入一个TabList控件，默认情况下，它显示成一个简单的黑色边框。在设计时的环境中它根本不象一个列表框或者其它什么东西。但在工程运行时，你就会看到它确实显示为一个列表框。到底怎么回事呢？

如果你为 TabList 选择了 OnDraw()函数，Control Wizard 会生成下面的代码：

```
void CTabListCtrl::OnDraw(CDC*pdc,const CRect & rcBounds,const CRect & rcInvalid)
 {
DoSuperclassPaint(pdc,rcBounds);
}
```

函数 DoSuperclassPaint()负责当需要重画时通知作为子类的列表框控件画出自身，但由于某些原因，在设计时的环境中它认为没有必要重画。这就是说，我们必须为设计时的环境增加作图代码。一般的 VB 列表框控件在设计时的环境中显示成一个白色方块，并将控件名字显示在左上角。我们可以模仿这种操作。下面是我们第一次编写的设计时的环境重画 OLE 控件的代码；

```
void CTabListCtrl::OnDraw(CDC*pdc,const CRect & rcBounds,const CRect & rcInvalid)
 {
if(AmbientUserMode());
  DoSuperclassPaint(pdc,rcBounds);
else
  {
  CBrush* OldBrush = pdc-> SelectObject ((HBRUSH) GetStockObject ( WHITE_
  BRUSH));
  pdc->Rectangle(rcBounds);
  pdc->TextOut(rcBounds.left,rcBounds.top,AmbientDisplayName());
  pdc->SelectObject(OldBrush);
  }
}
```

继承的方法 AmbientUserMode()当控件在被一个运行的应用程序使用时返回 TRUE，而在设计阶段使用时返回 FALSE。如果控件容器支持控件名属性的话(比如 Visual Basic 4.0就支持这个属性)，AmbientDisplayName()将返回控件的名字。

这两个函数是存取环境属性的两个例子。我在前面介绍过这些函数，但可能只有接触到它们的使用才会使你真正理解它们。由于 TabList 控件是嵌入容器中的 OLE 对象，所以容器可能会需要存取一些控件自身的属性；而控件也同样可能会需要存取容器的一些属性。这些属性就是环境属性(ambient property)。通常它们用于允许控件嵌入到容器中，而一些环境属性，如上面提到的两个，则另有目的。比如，根据控件是否使用在设计时的环境中而提供不同的显示或行为。在设计时的环境中，我们新控件的显示将如图16.10所示。

由于 OnDraw()方法中的设备环境已经选择了环境背景色的刷子(在设计时的环境中)，所以我们必须用一个库存刷子代替这个刷子。这个刷子以及它的颜色并不受 BackColor 库存属性值的影响。为了在设计时的环境中使界面更友好，我们必须显式生成以 BackColor 作为颜色的刷了。所以我们必须再次修改 OnDraw()方法：

```
void CTabListCtrl::OnDraw(CDC*pdc,const CRect& rcBounds,const CRect & rcInvalid)
 {
if(AmbientUserMode())
  DoSuperclassPaint(pdc,rcBounds);
```

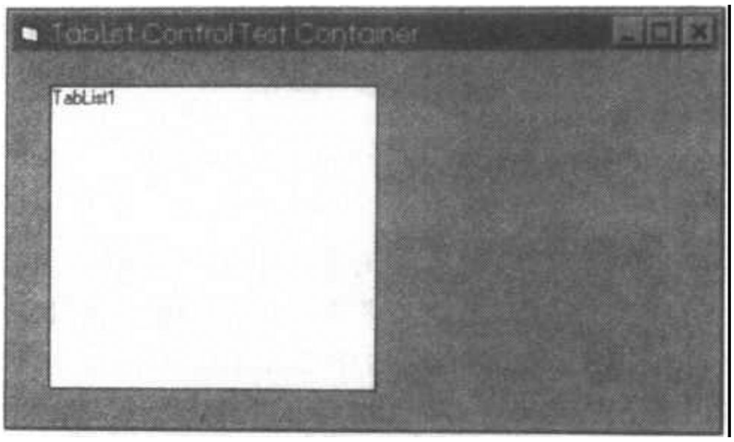

图16.10 设计时环境中的 TabList 控件显示

```
else
  {
  CBrush Brush(TranslateColor(GetBackColor()));
  CBrush * OldBrush=pdc->SelectObject(&Brush);
  pdc->Rectangle(rcBounds);
  pdc->SetTextColor(TranslatcColor(GctForcColor));
  pdc->TextOut(rcBounds.left rcBounds.top,AmbientDisplayName());
  pdc->SelectObject(OldBrush);
      }
}
```

这里之所以需要调用 TranslateColor()是因为 OLE 不使用 COLORREF。它使用的是一种叫做 OLE_COLOR 的未公开的数据类型。这种数据类型同 COLORREF 一样都是32位的，但它的内部格式更类似于 Visual Basic 的颜色表征。有意思的是，虽然提供了 TranslateColor()，但却没有一个将 OLE_COLOR 转换成 COLORREF 的函数！

在设计时的环境中，当设计人员改变控件背景颜色时，效果就会立即表现出来。可是控件在运行时的环境中这种改变则不会对控件的显示产生什么影响！这是因为内部代码并不能通知底层的列表框改变颜色。我们必须自己去做。

当一个子窗口要重画时，父窗口会接收到一个 WM_CTLCOLOR 消息。Class Wizard 会为这个消息提供下面的处理器：

```
HBRUSH CTabListCtrl::OnCtrlColor(CDC * pDC,CWnd * pWnd,UINT nCtlColor)
 {
//TODO: Add your mcssagc handlcr codc hcrc and/or call dcfault
return COleControl::OnCtlColor(pDC,pWnd,nCtlColor);
}
```

显然，上面实现代码并没有执行什么有用的操作。我们必须替换掉原来的操作，提供一个背景刷子并且改变文本颜色。这就是说，我们必须有一个背景刷子。即为每个增强的列表框控件类增加一个成员变量：

```
class CTabListCtrl: public COleControl
 {
```

```
        ⇓
        ⇓
//Implementation
protected:
~CTabListCtrl();
CBrush * Brush;
        ⇓
        ⇓
};
```

Brush 指针必须在构造函数中初始化为 NULL：

```
CTabListCtrl::CTabListCtrl(): Brush(NULL)
 {
InitializeIIDs(&IID_DTabList,&IID_DTabListEvents);
//TODO: Initialize your control' s instance data here.
}
```

16.5 选择一种颜色

在构造函数中我们并不生成刷子，因为当控件被构造时 BackColor 值还没确定。只有当 OnResetState()被调用时才确定了它的值，并且这个值可能并非你想要的。

OLE_COLOR 本质上就是 COLORREF，但具有一些额外的功能。我在第四章介绍过，COLORREF 是一个四字节结构，最低字节代表红值、第二字节代表绿值、第三字节代表蓝值、最高字节没有使用。但在 OLE_COLOR 中，最高字节被使用了。比如，如果这个字节的最高位为1，那么低位字节则是指向系统颜色表的索引。这个颜色表可以通过调用 GetSysColor() API 函数获得。

生成一个 OLE_COLOR 值并不困难，COLORREF 值可以直接赋值给它。这就是为什么 Microsoft 公司没有提供它们之间的转换函数：根本没必要。你也可以使用系统颜色，只要你将系统颜色值与 0x80000000取或操作。

所以，在生成刷子之前，我们必须首先提供给 BackColor 属性和 ForeColor 属性一个初始值。这里，我们想分别使用 COLOR_WINDOW 和 COLOR_WINDOWTEXT 来模拟 Windows 列表框控件的默认颜色：

```
void CTabListCtrl::OnResetState()
 {
//Invocation of base method supplied by Control Wizard
COleControl::OnResetState();
//Specific default property values set by you
SetBorderStyle(1);
SetBackColor(0x80000000|COLOR_WINDOW);
SetForeColor(0x80000000|COLOR_WINDOWTEXT);
delete Brush;
```

```
Brush=new CBrush(TranslateColor(GetBackColor()));
}
```

如果已经创建了一个刷子，首先要把它删除。(对 NULL 指针进行 delete 操作往往是安全的。)我们必须记住要在控件的析构函数中删除掉这个刷子：

```
CTabListCtrl::~CTabListCtrl()
{
delete Brush;
}
```

但重要的是，当 BackColor 属性被首次加载或被改变时，一定要先删除旧刷子、再生成新刷子。在控件的 DoPropExchange()成员函数执行过程中所有的属性都被加载。也就是说，用户属性是在这个函数中加载的，而库存属性是在基类的 DoPropExchange()函数中加载的。并且基类的 DoPropExchange()函数是先被调用的。所以在以上那些操作过后，你可以删除旧刷子，而以正确的颜色生成一个新刷子：

```
void CTabListCtrl::DoPropExchange(CPropExchange* pPX)
{
ExchangeVersion(pPX,MAKELONG(_wVerMinor,_wVerMajor));
COleControl::DoPropExchange(pPX);
//TODO: Call PX_functions for each persistent custom property.
delete Brush;
Brush=new CBrush(TranslateColor(GetBackColor()));
}
```

我们还需要对 BackColor 值的实时变化做出反应。由于它是个库存属性，所以并没有一个能直接改变它的值的函数可用。但却有一个我们可以重载的虚函数 OnBackColorChanged()。我们必须自己动手将这个函数的说明加入到头文件中：

```
class CTabListCtrl::public COleControl
{
        ⇓
        ⇓
//Implementation
protected:
  ~CTabListCtrl();
  CBrush* Brush;
  virtual void OnBackColorChanged(void);
        ⇓
        ⇓
};
```

并且我们还必须自己动手将这个函数加入到实现文件中，Class Wizard 帮不上忙：

```
void CTabListCtrl::OnBackColorChanged(void)
{
delete Brush;
```

```
Brush=new CBrush(TranslateColor(GetBackColor()));
}
```

在 class Wizard 将 OnCtlColor()处理器加入到类的说明和消息映射中后，我们就可以实现这个消息处理器了：

```
HBRUSH CTabListCtrl::OnCtlColor(CDC*pDC,CWnd*pWnd,UINT nCtlColor)
{
pDC->SetTextColor(TranslateColor(GetForeColor()));
retum(HBRUSH)Brush->m_hObject;
}
```

你认为这样就万事大吉，是吗?实际上还没完呢!不过你不用担心，我们所做的努力是不会白费的。

你知道，底层控件在需要时要向父窗口发送消息，而 WM_CTLCOLOR 则是其中之一。在 VBX 中，VBX 管理器负责管理包含控件的表窗口(form window)，并把这些消息“反射”到控件过程中。这样既干净利索，又有效。所以，OLE 设计者当然必须找到另外一种方法来完成这些操作。当一个 OLE 控件以标准控件作为子类(subclass)时，OLE 为这个控件提供了一个特殊反射窗口。从理论上讲，你可以监视并且任意处理发送到反射窗口的消息，如 WM_CTL-COLOR。

这样做虽然已经简化了操作，但因为一些容器为了减轻系统负担将来很可能会自己提供反射窗口，所以要增加一组新的消息映射以容纳那些反射的消息。这样就不必考虑到底是谁负责反射消息。Class Wizard 现在还没有这个功能，所以你必须自己编写这部分代码。(注意，要将这些代码放到 Class Wizard 注释以外)。首先你要将函数原型放在类的说明中：

```
class CTabListCtrl: public COleControl
{
        ⇓
        ⇓
//Implementation
protected:
    ~CTabListCtrl();
    CBrush*Brush;
    virtual void OnRackColorChanged(void);
    LRESULT OnOcmCuColor(WPARAM wParam,LPARAM lParam),
        ⇓
        ⇓
};
```

“Ocm”表示“OLE Command Message(OLE 命令消息)”。你必须在实现文件中的消息映射部分增加一个宏：

```
BEGIN_MESSAGE_MAP(CTabListCtrl,COleControl)
  //{{AFX_MSG_MAP(CTabListCtrl)
  ON_MESSAGE(OCM_COMMAND,OnOcmCommand)
  ON_WM_CTLCOLOR()
```

```
//}} AFX_MSG_MAP
ON_OLEVERB(AFX_IDS_VERB_PROPERTIES,OnProperties)
ON_MESSAGE(OCM CTLCOLOR,OnOcmCtlColor)
END MESSAGE MAP()
```

消息处理器有且只有两个参数 wParam 和 lParam，并且这些消息的处理不是任意的。在这些函数中我们要加入自己的代码。这里我将消息参数重新组织并调用适当的处理器：

```
LRESULT CTabListCtrl::OnOcmCtlColor(WPARAM wParam,LPARAM lParam)
{
return(long)(LPVOID)OnCtlColor(CDC::FromHandle((HDC)wParam),
                  CWnd::FromHandle((HWND)LOWORD(lParam)),
                  (int)HIWORD(lParam));
}
```

由于 OnCtlColor()需要 MFC 对象参数，如 CDC 和 CWnd。我们通过使用相应类的静态函数 FromHandle()来获取类的对象。这些函数返回暂时对象，所以我们不必去删除这些对象。

现在去编译和测试吧！你会看到在设计和运行时背景颜色和文本颜色都可以改变的矩形块。

16.6 使用库存字体属性

我们要加入新控件中的库存属性之一就是 Font(字体)属性。这个属性代替了所有的 VBX 字体属性，如 FontBold，FontName，FontSize 等等。

在有过处理 BackColor 和 ForeColor 属性的经验后，你可能会以为处理 Font 属性还会同样地麻烦。幸好不是这样。虽然控件也有一个 OnFontChanged()虚函数，但它的基类实现正是我们所想要的：向底层窗口发送 WM SETFONT 消息。并且只在运行时环境中才被调用，所以我们的任务变得简单了：只需在设计阶段要获得和使用那种字体。

我们需要使用 COleControl 类定义的成员函数 SelectStockFont()。这个函数以设备环境作为参数并把新字体选入该设备环境中。我们本章的 OnDraw()函数最后如下所示：

```
void CTabListCtrl::OnDraw(CDC*pdc const CRect &rcBounds,const CRect & rcInvalid)
{
if(AmbientUserMode())
  DoSuperclassPaint(pdc,rcBounds);
else
  {
  CBrush*OldBrush=pdc->SelectObject(Brush);
  pdc->Rectangle(rcBounds);
  CFont*OldFont=SelectStockFont(pdc);
  pdc->SetTextColor(TranslateColor(GetForeColor()));
  pdc->TextOut(rcBounds. left. rcBounds. top,AmbientDisplayName());
  pdc->SelectObject(OldBrush);
  pdc->SelectObject(OidFont);
  }
```

}

16.7 建立属性页

如果当你在 Visual Basic 中测试新控件时碰巧打开了与属性框中“(Custom)”相关联的对话框时，你就会看到如图16.11所示的属性页。

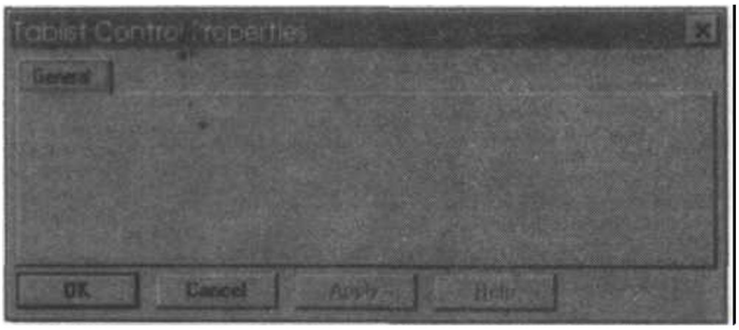

图16.11 一个空的属性页

每个控件都有一个类似的属性页。你的工作就是填写它。不要以为 Visual Basic 的属性框可以为你做这些事情，而你就可以跳过这部分工作了。其实编写 OCX 的目的是在容器而不是在 Visual Basic 中使用这些属性。否则，我们只需编写既快又易写的 VBX 了。属性页可能是将来所有的对象改变它们设计时的属性值的途径。所以如果 Visual Basic 将来的版本没有属性框，我也不会感到奇怪。

现在我们并不打算在普通页(general page)中填写东西。直到在下一章我们定义完我们的用户属性后再填写这个属性页。但我们可以为一些库存属性再增加两个属性页。我们可以增加三个属性页：一个是字体属性页，一个是颜色属性页，最后是图片属性页。不过，我们现在还不需要图片属性页，但我们确实定义了 Font 属性和两个颜色属性，所以要加入字体属性页和颜色属性页。

库存属性页是手工加入到每个控件实现文件中的结构中去的。这个结构当首次由 Class Wizard 生成时将如下所示：

```
BEGIN_PROPPAGEIDS(CTabListCtrl,1)
  PROPPAGEID(CTabListPropPage::guid)
END_PROPPAGEIDS(CTabListCtrl)
```

BEGIN_PROPPAGEIDS 宏的第一个参数是属性页的总数。当你增加一个库存属性页或用户属性页时，不要忘记增加这个参数值。加入两个属性页后，这个结构将如下所示：

```
BEGIN_PROPPAGEIDS(CTabListCtrl,1)
  PROPPAGEID(CTabListPropPage::guid)
  PROPPAGEID(CLSID_CColorPropPage)
  PROPPAGEID(CLSID_CfontPropPage)
END_PROPPAGEIDS(CTabListCtrl)
```

加入以上代码后，无需再做其它工作了。Fonts 属性页自动地连接到 Font 库存属性(以及其它字体属性)上，见图16.12。对于 Colors 属性页(见图16.13)，有一个好消息和一个坏消息。好消息是，象字体属性页一样，颜色属性页自动地连接到你拥有的全部颜色属性上。坏消息是

这是最丑陋的最低效的颜色选择对话框！

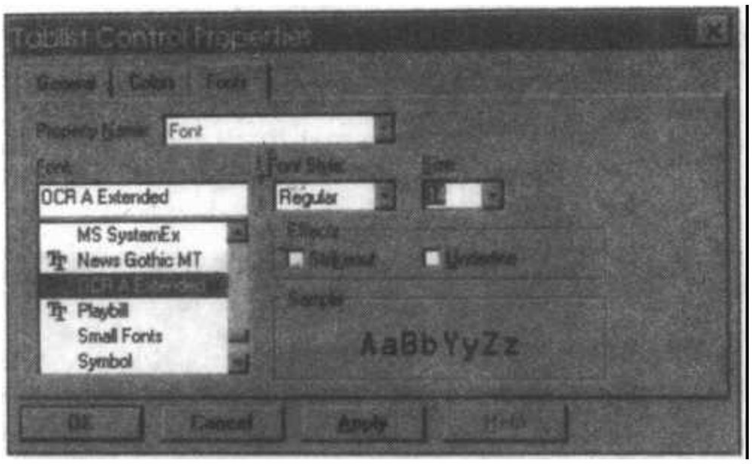

图16.12　Font 属性页

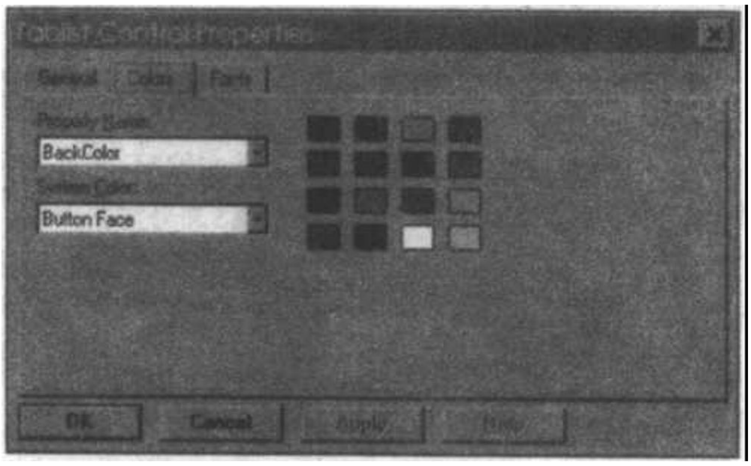

图16.13　Colors 属性页

16.8　继续前进

当前，我们的列表框还需要很多工作去做。我们在本章所加入的库存属性是非常有用的，但我们还需要一种获取和设置列表框数据的方法。需要增加用户属性和方法来完成这个工作。下一章我们将增加一些用户属性来改进列表框控件。我保证我们会做得越来越好！

第十七章　用户 OCX 属性

在上一章，我们将标准列表框从一般的平台移植成 OLE 控件。现在我们的工作是使这些被移植的列表框控件在新环境中能很好地存活。而我们所采用的方法就是增加用户属性。

17.1　介绍用户属性

对于 OCX 开发者来说，OCX 用户属性编写起来同编写 VBX 用户属性一样简单。如有可能，你应该重新使用从其它控件借过来的属性名(当然不一定非要这样)。但这种属性名重用并不能提供它们的实现。你必须自己去实现它们，这使你有机会尽可能发挥你的创造性；同时对终端用户想当然地胡乱使用这些属性你也要有相应的心理准备。

由于我们是以模拟列表框控件作为起始点的，我们应该看看它都有哪些属性可用。Visual Basic 的属性框提供了一系列的属性(见图17.1)。这个列表框所列的属性中有许多属性都是环境属性或库存属性。它这些属性从列表中删去后，剩下的都是必须实现的用户属性(见表17.1)。

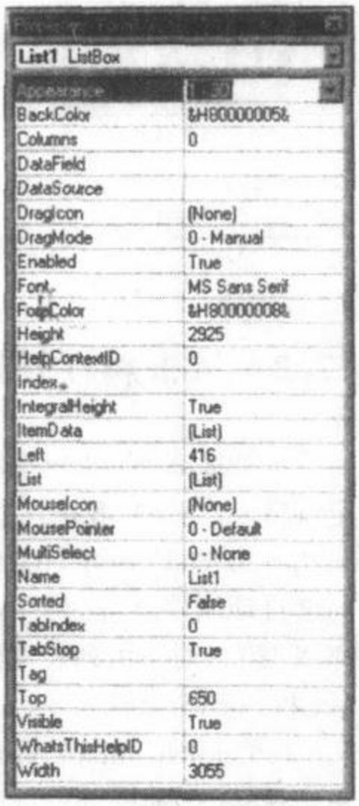

图17.1　列表框属性

显然，表17.1中所呈现的属性表并不是微不足道的。但这并不意味着需要做大量的工作，因为绝大多数属性实际上已经由列表框自己实现了。我们所要做的就是通过 OCX 将它们传递给低层代码。所以让我们从这些属性开始吧。

表17.1　需要实现的用户属性

属　性	描　　述
Appearance	提供两种不同作图方法的选择：平面的(最初 Windows 3.1风格)和三维立体的(流行的 Windows 95风格)。文档上说，这个属性对16位 OCX 是无效的
Columns	允许多列列表框。这个属性在 TabList 中的实现与在 ImageList 和 DrawList 中是不同的
DataField, DataSource	我们将不实现这两个数据敏感的属性
IntegralHeight	这个属性规定了列表框是否在垂直方向上自动调整大小以显示奇数个表项
ItemData	它是一个并列于 List 属性的数组。它允许你即使在由于排序、插入或删除前一项而发生了列表项的移动时，也可以将数据项连接到列表项上
List	是列表框的默认属性。通过这个数组可以存取其中的列表项
ListCount	只在运行时有效。这个属性反映了 List 数组中元素的个数
ListIndex	只在运行时有效。这个属性可以读取出来以确定 List 中哪个元素被选择了或编程写入它的值以设置当前的选择
MouseIcon, MousePointer	这两个属性允许设计者为列表框确定非标准鼠标的光标。我们将不支持这些不常用的属性

续表

属　性	描　　述
MultiSelect	规定了列表框中多选的三种水平：只单选、旧风格(Windows 21)的多选或扩展多选
NewIndex	只在设计时有效。它可以用来确定 List 中哪个元素是最近加入的
SelCount	这个属性主要应用于多选列表框，并只在运行时有效。它反映了当前选择项的个数
Selected	这个数组属性只在运行时有效。这个数组中存放着 List 中那些被选择的数据项
Sorted	规定列表框是否自动进行表项排序。
Text	这个属性等同于 List [ListIndex] 操作。它是只读的并且只在运行时有效
TopIndex	这个属性用于读取和设置 List 中最上端可见表项的索引。它只在运行时有效

17.2　从根属性开始

每种标准 Windows 控件都支持一定的属性。这些属性是当底层窗口被创建时通过包含一定风格来设置的。我把这些属性称为“根属性(root property)”。它们也叫做“pre-create”属性和“pre-hWnd”属性。在我们的列表框中，这些根属性包括列表框是否根据它所包含的表项的高度自动调整自身的大小、列表框中的表项是否自动排序等等。这些根属性特别容易应用于 OCX 接口。对于由列表框派生出来的控件来说，根属性包括 Columns、IntegralHeight、MultiSelect 和 Sorted。

IntegralHeight 和 Sorted 都是 BOOL 型。它们很容易被加入，因为它们可以作为成员变量来实现。这样 MFC 就可以为你管理这些值的获取和设置。当然，你可能想知道这些值是什么时候被设置的，所以 MFC 同时还会自动生成一个通知函数。

下一个属性 Columns 是 short 类型，它反映了列表框的列数。它的默认值是1。对于 ImageList 和 DrawList 控件来说，它也可以做为成员变量并且你现在就应加入它。但 TabList 控件要求它是一个“get/set”类属性，其中的原因我会在后面进行阐述。

同前一章中的库存属性一样，这些属性也是用 Class Wizard 加入的。这需要点击 OLE Automation 标签。图17.2显示了被加入的 Sorted 属性。注意，成员变量名并没有使用 m_Sorted，而是直接使用 Sorted，因为我讨厌 m_ 前缀。

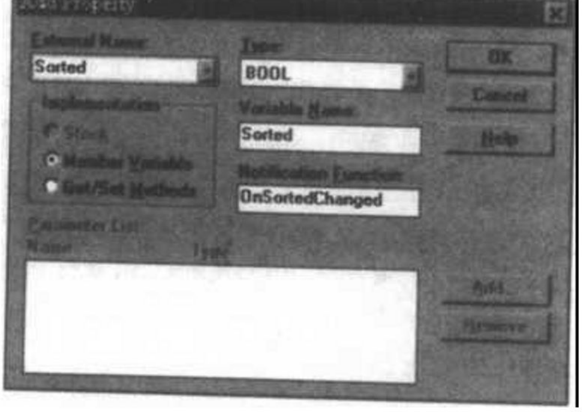

图17.2　加入 Sorted 属性

在加入这些属性后，它们的代码文件也被修改了。首先是两个变量和两个函数以及属性的发送 ID(Dispatch ID)被加入到了类的说明中：

```
class CTabListCtrl: public COleControl
  {
            ⇓
            ⇓
//Dispatch maps
// { {AFX DISPATCH(CTabListCtrl)
BOOL Integrall leight;
afx_msg void OnIntegralHeightChanged();
BOOL Sorted;
afx_msg void OnSortedChanged();
//}} AFX_DISPATCH
DECLARE_DISPATCH_MAP()
afx msg void AboutBox();
            ⇓
            ⇓
//Dispatch and event Ids
public:
  enum {
// { {AFX_DISP_ID(CTabListCtrl)
dispidIntegralHeight=1L,
dispidSorted=2L,
//}} AFX_DISP_ID
  };
};
```

其它的修改是在类的实现文件中。我们先来看看发送映射：

```
BEGIN_DISPATCH_MAP(CTabListCtrl,COleControl)
    // { {AFX_DISPATCH_MAP(CTabListCtrl)
    DISP_PROPERTY_NOTIFY(CTabListCtrl,"Integrall leight". IntegmIl leight
        OnIntegralHeightChanged,VT_BOOL)
     DISP_ PROPERTY_ NOTHY (CTabListCtrl, "Sorted", sorted OnSortodChanged,
VT_,BOOL)
    DISP_STOCKPROP_BORDERSTYLE()
    DISP_STOCKPROP_BACKCOLOR()
    DISP_STOCKPROP_ENABLED()
    DISP_STOCKPROP_FONT()
    DISP_STOCKPROP_FORECOLOR()
    DISP_STOCKPROP_HWND()
    //}} AFX_DISAPTCH_MAP
```

DISP_FUNCTION_ID(CTabListCtrl," AboutBox",DISPID_ABOUTBOX,AboutBox,
VT_EMPTY,VTS_NONE)
END_DISPATCH_MAP()

DISP_PROPERTY_NOTIFY 宏描述了属性名、存贮属性的数据成员名、通知函数名以及数据类型。

这些变量在函数 DoPropExchange()中既可以被设置成默认的值，也可以进行序列化(serialize)。奇怪的是,Class Wizard 并不生成这些代码。当然,编写序列化代码是很简单的事。有一组函数可以供你使用,这组函数都以 PX_开头,后面接着数据类型名。下面呈现的是 DrawList 控件的这部分代码，从中你可以看到 Columns 属性也被序列化了：

```
void CDrawListCtrl::DoPropExchange(CPropExchange * pPX)
{
Exchange Version(pPX,MAKELONG(wVerMinor,_wVerMajor));
COleControl::DoPropExchange(pPX);
PX_Bool(pPX,"IntegralHeight",mtegralHeight,TRUE);
PX_Bool(pPX,"Sorted",Sorted,FALSE);
PX_Short(pPX,"Colurmns",Columns,1);
delets Brush;
Brush=new CBrush(TranslateColor(GetBackColor()));
}
```

所有这三个属性一定要对 Windows 列表框的生成起作用。这需要在 PreCreateWindow()函数中修改窗口风格标志：

```
BOOL CDrawListCtrl::PreCreate Window(CREATESTRUCT &. cs)
{
cs. lpszClass=T("LISTBOX");
ifColumns>1()cs. stylel=LBS_MULTICOLUMN;
if(!IntegralHeight)cs slyle 1=LBS_NOINTEGRALHEIGHT;
if(Sorted)cs style=LBS_SORT;
return COlcControl::ProCreate Window(cs);
}
```

这些属性甚至可以在运行时被修改。如果修改了这些属性,原来的控件必须被撤消并以新属性值重新生成控件。我们要确保这些操作都发生在通知函数中：

```
void CTabListCtrl: OnIntegralHeightChanged()
{.
Recreate ControlWindow();
SetModifiedFlag();
}
void CTabListCtrl::OnSortedChanged()
{
RecreateControlWindow();
```

```
SetModifiedFlag();
}
```

17.3 实现枚举属性

第四个根属性 MultiSelect 的使用要比前面三个属性稍微麻烦一些。因为即使属性值是以枚举型数值(通常是0,1,2等)存贮的，它们呈现给用户的仍然是字符串："Single"、"Multiple"和 "Extended"。

在 VBX 控件中那些叫做"枚举属性"的属性是很简单的。当你定义这类属性时，你所做的就是提供相应枚举值列表。但在 OLE 控件中这变得复杂得多。最讨厌的是很难找到有关的文档资料。

在 OLE 控件中，枚举属性实际上是由与其它属性相同的数据类型组成的。所不同的是你必须提供代表枚举值的字符串。如果你的控件中有枚举属性，你必须重载从 COleControl 继承的三个函数。Class Wizard 并不能帮助你做这件事情，你必须自己输入函数原型。这三个函数是 OnGetPredefinedStrings()、OnGetPredefinedValue()和 OnGetDisplayString()；

```
class CTabListCtrl: public COleControl
 {
          ⇓
          ⇓
//Dispatch maps
    // { {AFX_DISPATCH(CTabListCtrl)
    BOOL, integralHeight;
    afx_msg void OnIntegralHeightChanged();
    BOOL Sorted;
    afx_msg void OnSortedChanged();
    short MultiSelect,
    afx_msg void OnMultiSelectChanged();
    //}} AFX_DISPATCH
    DECLARE_DISPATCH_MAP();
    virtluai BOOL OnGetPredefinedStrings(DISPID DispID,CStringArray*Strings,
              CDWordArray*Cookies);
    virtual BOOL OnGetPredefinedValue(DISPID DispID,DWORD Cookie,
              VARIANTFAR* Value);
    virtual BOOL OnGetDisplayString(DISPID DispID CString &. String);
          ⇓
          ⇓
//Dispatch and event Ids
pubiic:
  enum {
    // { {AFX DISP ID(CTabListCtrl)
```

```
    dispidIntegralHeight = 1L,
    dispidSorted = 2L,
    dispidMultiSelect = 3L,
    //}} AFX_DISP_ID
    };
};
```

这三个函数都以一个“dispatch ID”作为它的第一个参数。在上面那段代码中你可以看到类中所定义的发送 ID，并且这三个函数使用它们来识别所要操作的属性。这三个函数被设计成既能处理枚举属性又能处理非枚举属性。如果所要操作的是非枚举属性，则返回 FALSE。

当系统需要某种属性的全部枚举值时则要调用 OnGetPredefinedString()。你要提供一个存贮字符串指针数组的缓冲区。这个字符串数组并不是个带有索引的数组(即第二个元素就用2来表示)，所以你必须还要填写另外一个数组 Cookies，使第二个字符串与第二个“Cookies”相关联(即 cookies 数组中的元素值是字符串数组的索引值)。

为了实现这个函数，字符串应该存贮在字符串表资源中。我所加入的字符串见图17.3。当然我们必须将包含这三个字符串的数据成员加入到类的说明中。为了避免命名上的冲突，我把它们放进了一个结构中：

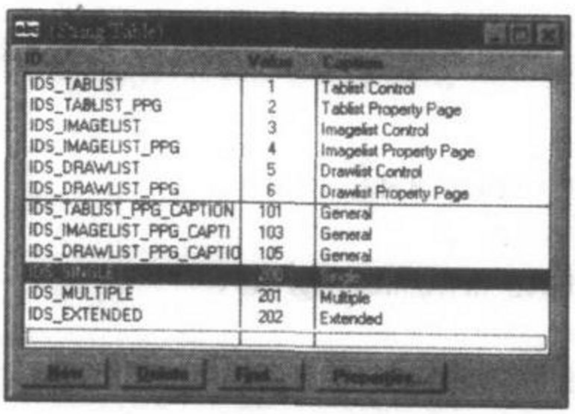

图17.3 字符串表中的枚举值

```
class CTabListCtrl: public COleControl
 {
          ⇓
          ⇓
struct tag_MultiSelect_Values
      {
    CString Single;
    CString Multiple;
    CString Extended;
          } MultiSelect_Value;
          ⇓
          ⇓
};
```

MFC 的 CString 类虽然没有以字符串表作为参数的构造函数，但它却有一个 LoadString

()方法可以调用：

```
CTabListCtrl::CTabListCtrl(): Brush(NULL)
{
InitializeIIDs(&IID_DTabList,&IID_DTabListEvents);
MultiSelect_Values. Smgle. LoadString(IDS_SINGLE);
MultiSelect_Values. Multiple. LoadString(IDS_MULTIPLE);
MultiSelect Values. Extended LoadString(IDS_EXTENDED);
}
```

下面是 OnGetPredefinedString()函数的定义：

```
BOOL, CTabListCtrl::OnGetPredefinedStrings(DISPID DispID,CStringArray * Strings,
        CDWordArray * Cookies)
{
switch(DispID)
   {
   case dispidMultiSelect:
   Strings->Add(MultiSelect Values. Single);
   Cookies->Add(0);
   Strings->Add(MultiSelect_Values. Multiple);
   Cookies->Add(1)
   Strings->Add(MultiSelect_Values. Extended);
   Cookies->Add(2);
   return TRUE;
   }
return FALSE;
}
```

正如前面所述，这个函数除了 DispID 是 dispidMultiSelect 外，这个函数返回 FALSE。这里有三个枚举字符串被加入到了 Strings 数组中，而与之相关联的数值以同一顺序被加入到了 Cookies 数组中。

只有当需要全部的枚举列表时才调用这个函数。比如，当所有可能的属性值需要显示在属性框中时，就要调用这个函数。当用户选择其中的一个值时，OnGetPredefinedValue()会被调用：

```
BOOL CTabListCtrl::OnGetPredefinedValue(DISPID DispID, DWORD Cookie,
        VARIANTFAR * Value)
{
switch(DispID)
   {
   case dispidMultiSelect:
      Value->vt=VT_12;
      Value->iVal=(short)Cookie;
      return TRUE;
```

```
    }
return FALSE;
}
```

Valuc 参数是一个指向 VARIANT 的指针。该数据类型是一种联合(union)。VisualC++ 1.52c 中没有这种数据类型的联机帮助，下面是它的结构：

```
typedef struct FARSTRUCT tagVARIANT VARIANT;
typedef struct FARSTRUCT tagVARIANT VARIANTARG;
typedef struct tagVARIANT {
VARTYPE vt;
unsigned short wReservedl;
unsigned short wRescrved2;
unsigned short wReserved3;
union {
  unsigned char bVal;
  short iVal;
  long IVal;
  float HtVal;
  double dblVal;
  VARIANT_BOOL bool;
  SCODE scode;
  CY cyVal;
  DATE date;
  BSTR bstrVal;
  IUnknown FAR*punkVal;
  IDispatch FAR*pdispVal;
  SAFEARRAY FAR*parray;
  unsigned char FAR*pbVal;
  short FAR*piVal;
  long FAR*plVal;
  float FAR*pfltVal;
  double FAR*pbdlVal;
  VARIANT_BOOL FAR*pbool;
  SCODE FAR*pscode;
  CY FAR*pcyVal;
  CATE FAR*pdatc;
  BSTR FAR*pbstrVal;
  IUnknown FAR*FAR*ppunkVal;
  IDispatch FAR*FAR*ppdispVal;
  SAFEARRAY FAR*FAR*parray;
  VARIANT FAR*pvarVal;
```

```
void FAR *byref;
   };
};
```

这个数据结构吓了我一跳。我想它唯一没有定义的只剩下指向狄更斯小说的指针了吧!你必须设置 VARTYPE 元素 vt 的值以表明你发送的是什么。对于 short 类型的 MultiSelect 属性来说，我们对 iVal 成员进行赋值并通过将 VT_12赋值给 vt 成员来表明数据类型。当然所有的数据类型都有相应的定义。

这三个函数中最后一个是 OnGetDisplayString()。当系统需要知道属性的当前值并希望以适当的枚举值来显示这个值时需要调用这个函数：

```
BOOL CTabListCtrl::OnGetDisplayString(DISPID DispID,CString & String)
 {
switch(DispID)
  {
  case dispidMultiSelect:
    switch(MultiSelect)
     {
      case 0:
        String=MultiSelect_Values. Single;
        break;
      case1:
        String=MultiSelect_Values. Multiple;
        break;
      case2:
        String=MultiSelect_Values. Extended;
        break;
               }
      return TRUE;
  }
return FALSE;
}
```

在上面这个函数中我们完成 MultiSelect 属性的内部管理，而 MultiSelect 的通知函数与前面三个通知函数类似：

```
void CTabListCtrl::OnMultiSelectChanged()
 {
RecreateControlWindow();
SetModifiedFlag();
}
```

在 PreCreateWindow()函数中增加的代码也与其它属性类似：

```
BOOL CTabListCtrl::PreCreateWindow(CREATESTRUCT & cs)
 {
```

```
cs. lpszClass=_T("LISTBOX");
if(IIntegralHeight(cs.style|=LBS_NOINTEGRALHEIGHT
if(Sorted)cs. style|=LBS_SORT;
if(MultiSelect==1)cs. style|=LBS_MULTIPLESEL;
else if(MultiSelect=2)cs. style|=LBS_EXTENDEDSLL;
return COleControl::PreCreateWindow(cs);
}
```

最后，我们必须在 DoPropExchange()中加入一项：

```
void CTabListCtrl::DoPropExchange(CPropExchange* pPX)
{
Exchange Version(pPX,MAKELONG(wVerMinor,wVerMajor));
COleControl::DoPropExchange(pPX);
PX_Bool(pPX,"IntegralHeight",IntegralHeight,TRUE);
PX_Bool(pPX,"Sorted",Sorted,FALSE);
PX_Short(pPX,"MultiSelect",MultiSelect,0),
delete Brush;
Brush=new CBrush(TranslateColor(GetBackColor()));
}
```

由于你已经增加了属性，所以在编译之前，不要忘记运行 Tool…Make Typelib 命令。一旦你编译过了，运行 Visual Basic 后你就应该看到那些新属性，包括 MultiSelect 属性所对应的枚举值。

17.4 瞄准底层控件的属性

下面我们要实现的属性是那些已经被底层 Windows 列表框控件实现的属性。为了使它们发挥作用，我们只需向控件发送适当的消息。在这一部分，我们要实现三个属性：ListCount、ListIndex 和 TopIndex。这三个属性的数据类型都是 short 型，并且都不会在设计时环境中出现在属性框中。

这种类型的属性是使用 Get/Set 函数来实现的，这是因为 OLE 控件本身并不需要将这些数据作为自己的数据成员来存贮，其实这样做也没什么用。相反，OLE 控件只是作为一种中介，一个信息传输的通道。(译者注：即在需要时控件会向底层列表框发送相应的消息来读取或写入这些数据。)图 17.4 显示了当 Class Wizard 增加其中一个属性时的增加属性(Add Property)对话框。注意，我已经删掉了 Set 函数，因为这个属性是只读的。当然，如果属性是只写的，则需要删除 Get 函数。

这三个属性值都不是永久的，所以我们不必向 DoPropExchange()函数增加任何代码。我们所要做的就是在每个 Get/Set 函数中向底层窗口发送消息。发送消息再简单不过了。你知道，你的新 OLE 控件类是由 COleControl 类派生出来的，而这个基类最终是由 CWnd 类派生出来的。CWnd 类有一个 SendMessage()成员可以用来向它自身发送信息。如果你不指定它的 wParam 和 lParam 参数，它们默认为0。所以，比如要实现 GetListCount()，我们只需要一行代码：

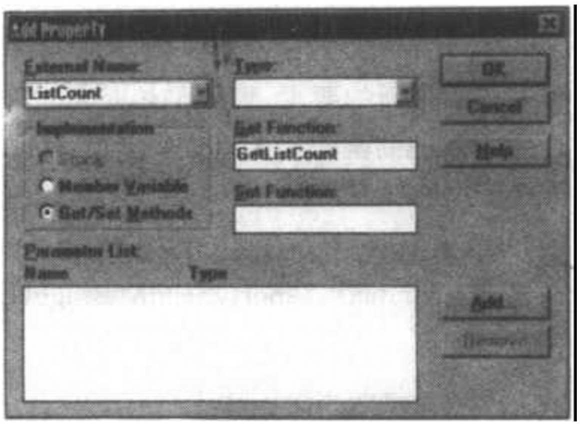

图17.4 增加只读 Get/Set 属性

```
short CTabListCtrl::GetListCount()
  {
  if(AmbientUserMode())
      return(short)SendMessage(LB_GETCOUNT);
  else
    {
    ThrowError(CTL_E_PERMISSIONDENIED,IDS_NOSHOW);
    return NULL;
    }
  }
```

AmbientUserMode()当控件工作在运行时应用程序中时返回 TRUE；而当一个表处于设计状态时，返回 FALSE。对于后者为了防止属性出现在容器的属性框中，在 Get 函数中只是简单地进行出错处理。

进行出错处理的 COleControl::ThrowError()函数使用标准的异常处理。这个函数只应在 Get/Set 属性值出错时被使用。其它情况下你可以激发库存 Error 事件。ThrowError()带有两个参数。第一参数是字符串缓冲区，而第二个参数是字符串表的 ID。即使我们提供了 IDS_NOSHOW 所标识的错误字符串，这个字符串对用户来说也是不可见的，用户不会感觉到有错误发生。

当系统在设计阶段试图设置这个属性值时，就会有错误发生。Visual Basic 并不在乎它是什么错，所以只读属性从不会出现在属性框中。在运行时，程序试图对这个属性赋值将会引发“不支持设置(set not supported)”错。

TopIndex 又是一个只有在运行时才可以读取的属性：

```
short CTabListCtrl::GetTopIndex()
  {
  if(ArnbientUserMode())
    return(short)SendMessage(LB_GETTOPINDEX);
  else
      {
    ThrowError(CTL_E_PERMISSIONDENIED,IDS_NOSHOW);
    return NULL;
```

```
    }
}
```

在设置 TopIndex 值时我们有一个新的要求，即我们绝对不能允许容器设置一个超出范围的非法索引。所以我增加了一个小的辅助函数：

```
BOOL CTabListCtrl::Is ValidIndex(short TestIndex)
 {
return((TestIndex>-1)&&(TestIndex<(short)SendMessage(LB_GETCOUNT)));
}
```

有了这个函数，SetTopIndex()编写起来就简单了：

```
void CTabListCtrl::SetTopIndex(short New Value)
 {
if(IsValidIndex(New Value))
    {
   SendMessage(LB_SETTOPINDEX,New Value);
   SetModifiedFlag();
    }
else
          ThrowError ( CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
}
```

对 ListIndex 的处理稍微复杂一些，倒不是因为 OLE，而是因为具有单选风格列表框与多选风格的列表框在操作上有些不同。对于单选列表框来说，如果有个表项被选择了，ListIndex 返回那个表项的索引(如果没有表项被选择，则返回1)。而在多选列表框中，ListIndex 返回具有输入焦点框的表项索引，而不管这个表项是否处于被选择状态：

```
short CTabListCtrl::GetListIndex()
 {
if(AmbientUserMode())
 {
if(MultiSelect)return (short)SendMessage(LB_GETCARETINDEX);
else return (short)SendMessage(LB_GETCURSEL);
}
else
    {
   ThrowError(CTL_E_PERMISSIONDENIED,IDS_NOSHOW);
   return NULL;
    }
}
void CTabListCtrl::SetListIndex(short New Value)
 {
if(IsValidIndex(NewValue)) ||((MultiSelect)&&NewValue==-1))
```

```
    {
    if(MultiSelect)SendMessage(LB_SETCARETINDEX,New Value);
    else SendMessage(LBSETCURSEL,NewValue);
    SetModifiedFlag();
    }
else
        ThrowError ( CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE),
}
```

17.5 种子属性

有一些属性，它们的值是由控件的一些行为设置的，并且是只读的，不能被写入——就象 DNA 一旦植入种子就永远无法再改变一样。严格地说，ListCount 属性就是这种属性。但这个属性的内部维护并不需要我们的介入。可 NewIndex 属性则不是这样。

列表框本身并没有 NewIndex 属性。但无论何时当向列表框加入(Add)或插入(Insert)一个表项时，完成操作的消息返回新表项的索引。由于设置属性值时函数没有返回值，所以我们必须保存这个索引。容器也就可以读取 NewIndex 这个属性了。

当我们向控件增加 NewIndex 属性时，我们必须把它设置成 Get/Set 属性。这是因为 OLE 并不能维护它。但我们仍需要将这个属性作为类的数据成员，所以我们加入它的定义：

```
class CTabListCtrl: public COleControl
  {
/
/
private:
   short NewIndex;



};
```

这个属性的默认值是－1。并且它不是个持久(persistent)的属性，所以不必为它再修改 DoPropExchange()，但必须在函数 OnResetState()中设置它的默认值：

```
void CTabListCtrl::OnResetState()
  {
/
/
NewIndex=-1,
}
```

如果它不是个“no show(不可见)”属性，那么在 GetNewIndex()中返回它的值只需一行代码：

```
short CTabListCtrl::GetNewIndex()
```

```
  {
if(AmbientUserMode());
    return NewIndex;
else
   {
  ThrowError(CTL_E_PERMISSIONDENIED,IDS_NOSHOW);
  return NULL;
  }
}
```

NewIndex 是只读属性，所以没有 Set 函数。

17.6 加入 text 属性

下一个我们要实现的属性是 Text 属性。当你使用 Class Wizard 增加 Text 属性时，它允许你将这个属性作为库存属性。通常，这是你所希望的。但库存 Text 属性只用于存放窗口标题，而列表框是不需要窗口标题的。所以我们必须将这个属性作为用户属性来实现，更确切地说是 Get/Set 属性。并且由于 Text 属性是只读的，所以你必须删除 Set 函数。

无论是作为库存属性还是用户属性，Text 属性的数据类型都是 BSTR。幸好 MFC 的 CString 类已经提供了向这种数据类型转换的函数。实际上，Class Wizard 的框架代码为我们准备了一个 CString 变量，依靠它返回正确的值。你所要做的就是在字符串中放入有意义的内容。

在这里，“有意义的内容”就是列表框中当前被选择的表项(对于单选列表框)或最后一次被单击的表项(对于多选列表框)的文本：

```
BSTR CTabListCtrl::GetText()
  {
if(AmbientUserMode())
  {
shorti=GetListIndex();
CString s;
if(i>-1)
   {
  short Length-(short)SendMessage(LB_GETTEXTLEN,(WPARAM)i);
   SendMessage(LB_GETTEXT,(WPARAM)i, (LPARAM)(LPSTR)s. GetBuffer
(Length));
  s. ReleaseBuffer(Length);
  }
return s. AllocSysString();
}
else
   {
```

```
ThrowError(CTL_E_PERMISSIONDENIED,IDS_NOSHOW);
return NULL;
}
}
```

SelCount 属性返回的是列表框中当前被选表项的数目。它通常是只读属性。但在 OCX 中我们可以允许将它赋值为0(以删除所有的表项)，但如果将它赋值为其它值时要进行出错处理：

```
short CTabListCtrl::GetSelCount()
{
if(AmbientUserMode())
  {
  if(MultiSelect)return(short)SendMessage(LB_GETSELCOUNT);
  else return(short)(SendMessage(LB_GETCURSEL)>-1)?1:0;
  }
else
  {
  ThrowError(CTL_E_PERMISSIONDENIED. IDS_NOSHOW);
  return NULL;
  }
}
void CTabListCtrl::SetSelCount(short New Value)
{
if(New Value==0)
  {
  if(MultiSelect)SendMessage(LB_SETSEL,FALSE,MAKELPARAM(-1,0));
  else SendMessage(LB_SETCURSEL,(WPARAM)-1);
  SetModifiedFlag();
  }
  else ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
  }
```

17.7 实现数组类属性

现在还有三个数组类属性需要实现。VBX 对数组类属性的处理是很糟糕的。好在 OLE 控件已经很好地解决了这个问题。对 OLE 控件来说，它只是下面这个问题的一个方面：通过参数来存取属性。

你想想看，在程序中，一维数组元素的获取(译者注：如 Data[i])是不是非常象调用带有一个参数的方法?同样的逻辑，为什么不把它们看成是一样的呢?这样做的另一个好处是你可以生成多维数组。VBX 对多维数组的处理更显得软弱无力了。

我们从 List 属性开始。通过这个属性我们可以存取列表框中的各个表项。你仍需要使用 Class Wizard，但当增加新属性时，你要点击 Parameter List 列表框旁边的 Add…按钮。它会弹出一个增加属性参数(Add Property Parameter)对话框(见图17.5)。

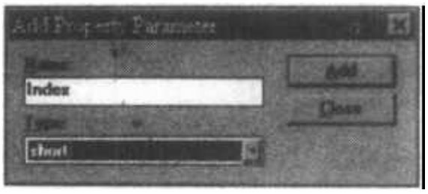

图17.5　增加属性参数

当 Class Wizard 生成 Set 和 Get 函数时，它为每个函数都包含了一个 short 类型的 Index 参数：

```
BSTR CTabListCtrl::GetList(short Index)
 {
if(IsValidIndex(Index)||Index==-1)
   {
  CString s;
  if(Index>-1)
      {
    short Length=(short)SendMessage(LB_GETTEXTLEN,(WPARAM)Index);
     SendMessage(LB_GETTEXT,(WPARAM)Index, (LPARAM)(LPSTR)s. Get-
Buffer(Length));
    s. ReleaseBuffer(Length);
    }
  reurn s. AllocSysString();
  }
else
  {
      ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
  return NULL;
  }
}
```

注意，我们在这里并没有调用 AmbientUserMode()。你可能并不希望 List 属性出现在 Visual Basic3. 0的属性框中，但即使出现了也无所谓。因为当列表框中没有表项时任何获取表项的操作都会引发“Invalid property value(无效属性值)”错，并且会使属性不会出现在属性框中。

设置 List 属性意味着替换掉已经存在的表项，可底层列表框控件并不支持这种操作。但我们仍要这么做。我们先存贮相关索引的表项数据，接着删除该表项，再在相同位置插入新表项，最后恢复该表项数据。注意，这个函数也重设了 NewIndex 的属性值：

```
void CTabListCtrl::SetList(short Index,LPCTSTR)
```

```
  {
  if(IsValidIndex(Index))
    {
    DWORD ItemData=SendMessage(LB_GETITEMDATA,(WPARAM)Index);
    SendMessage(LB_DELETESTRING,(WPARAM)Index);
    NewIndex=(short)SendMessage(LB_INSERTSTRING,(WPARAM)Index,
          (LPARAM)NewIndex);
    SendMessage(LB_SETITEMDATA,(WPARAM)NewIndex, ItemData);
    SetModifiedFiag();
    }
    else  ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
  }
```

说到表项数据，它是通过控件的 ItemData 数组属性来存取的：

```
long CTabListCtrl::GetItemData(short Index)
  {
  if(IsValidIndex(Index))
  return SendMessage(LB_GETITEMDATA,(WPARAM)Index);
  else
    {
        ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
    return NULL;
    }
  }
void CTabListCtrl::SetItemData(short Index,long New Value)
  {
  if(Is ValidIndex(Index))
    {
    SendMessage(LB_SETITEMDATA,(WPARAM)Index, NewValue);
    SetModifiedFlag();
    }
  else
        ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
    }
```

只剩下最后一个属性 Selected 了。只有多选风格的列表框直接实现了这个属性。所以我们还需要增加一些代码来处理多选和单选列表框在这一属性实现上的不同：

```
BOOL CTabListCtrl::GetSelected(short Index)
  {
```

```
if(Is ValidIndex(Index))
if(MultiSelect)return(BOOL)SendMessage(LB_GETSEL,(WPARAM)Index);
else return(BOOL)(SendMessage(LB_GETCURSEL)==Index);
else
    {
        ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
      return NULL;
      }
  }
```

在多选列表框中，确定某个表项是否是处于当前选择状态是 LB_GETSEL 消息的任务。在单选列表框中，你必须找到当前选择项并与 Index 参数进行比较。

对 Selected 属性进行赋值要复杂一点，因为这个操作既可能选择某一项也可能取消对某一项的选择：

```
vodi CTabListCtrL::SetSelected(short Index,BOOL New Value)
 {
if(IsValidIndex(Index))
    {
    if(MultiSelect)SendMessage(LB_SETSEL,(WPARAM)New Value, MAKELPARAM
(Index,0));
    else if(New Value)SendMessage(LB_SETCURSEL,(WPARAM)Index);
    else SendMessage(LB_SETCURSEL,(WPARAM)-1);
    SetModifiedFlag();
      }
else
      ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
   }
```

使用带-1索引值的 LB_SETCURSET 消息的目的就是取消对当前选择项的选择。

17.8 你说你想派生一个控件？

我们并没有为 TabList 控件实现 Columns 属性，这三个控件都没实现该属性。实现这个属性实在是件费时费力的事。

在下一章我们将要为每个控件增加一些各自的特色。可你一定会问：为什么我们非要这么做？难道不是用 C++进行面向对象的程序设计吗？如果是，那么为什么我不能只编写一个一般的列表框控件类，然后再从该类中派生出这三个实际的控件呢？

的确，你是可以这样做。但要完成这些工作需要很多步骤，因为这种情况下 Control Wizard 和 Class Wizard 都帮不上忙。

首先你需要在你基类的实现文件中加入 BEGIN_MESSAGE_MAP 宏，并删除下面的映

射项：

ON_OLEVERB(AXS_IDS_VERB_PROPERTIES,OnProperties)

如果你不这么做，容器会列出你的派生类和基类的属性，也就是说，继承的属性会列出二次。

下一步，如果基类不能被它自身来使用(即它是个“抽象”类)，那么你就应该改变它的 UpdateRegistry()函数，简单地返回 TRUE。这可以避免基类被注册，进而出现在 OCX 的控件列表中。同时，你必须对基类的属性页注册做同样的处理。

在基类的构造函数中，你需要删除对 InitializeIIDs()的调用。这个函数为基类读入类型信息。如果不删除这个函数，当你的 OCX 被下载时将会发生内存遗漏，因为读入的类型信息没被释放。

你还要扎入到工程的. ODL 文件中(要特别小心)为基类加入必要的解释信息。ODL 文件是工程描述文件。无论你什么时候运行 Tool.. Make Typelib 命令时，这个文件都要被修改。而这个命令是在你增加或删除属性、事件或方法后必须首先要运行的。所以每次你都要检查 ODL 文件以确保你的解释信息没被篡改。

你还要找到基类的 DoPropExchange()成员函数并删除下面这一行代码：

ExchangeVersion(pPx,MAKELONG(_wVerMinor,_wVerMajor));

这是因为你的派生类会调用这个函数，这样这个函数就会被调用两次。

对于 DoPropExchange()函数，你要在派生类中找到该函数并改变这一行代码：COleControl:: DoPropExchange(pPX)；以调用父类而不是祖父类的 DoPropExchange()函数。

这些修改代码的工作还算是容易的，更困难的是对 ODL 文件的修改。你必须手工更新属性、事件以及方法的发送 ID。这可不是件微不足道的任务，因为发送 ID 不是简单的数字。它基本上分为两部分。它的低字是基于1的与发送映射(dispatch map)顶部的距离；而高字则是基于0的与最高层派生类的映射距离。更糟糕的是，事件 DISPID 的处理与属性和方法的又是不同的。你可以参考 MFC 的 Tecluncal Note＃39，它详细描述了如何计算出那些发送 ID。

我想说的是，你最好不要这么做！

17.9 Appearance(外观)属性

还有一个属性我们不打算实现，它就是 Appearance 属性。它是个32位 OLE 控件的库存属性，不能在16位控件中使用。可为什么我们不手工地实现它呢？

如果我们编写的不是从标准 Windows 控件派生出来的控件的话，我们会很容易手工地实现它。我们可以放弃 BorderStyle 属性而用 Appearance 属性来代替它。Appearance 属性是枚举类属性，带有两种选择：“Flat(平面)”和“3-D(立体)”。根据所做的选择，我们只需画出简单地矩形框或带有立体阴影的边框。

可问题是我们的控件是从控件类派生出来的，根本没有用来画立体阴影的地方。OLE 控件不允许我们在控件边界外部进行作图，并且规定控件大小的任务是由 OLE 子系统来完成的。我们所使用的边界是子类的边界。

你可以用另外一种方法：不从控件类进行派生，而是将控件类窗口作为本控件的一个子窗口。这样你就可以任意控制子窗口的大小了，包括调整子窗口的大小使之比 OLE 控制小一点以便留出画边框的空间。这种情况下，由于 OLE 控件负责画出边框(平面的或立体的)，所以

你需要创建一个不带边框的子窗口。

你可以将子窗口类的对象(这里是 CListBox)作为控件的一个属性。这样就无需使用 SendMessage()函数，而是使用该类的对象方法进行更容易地操作。并且来自子窗口的消息会直接发送到 OLE 控件窗口，我们只需设置相应的消息处理映射就可以了。

但这样做所带来的负担就是系统中需要多维护一个窗口。你要自己决定是否真的加入 Appearanee 属性。

在下一章，我们将向 TabList、ImageList 和 DrawList 中增加一些反映它们特色的属性。我之所以将一部分任务放在本章阐述，是因为这些操作细节更多地包含了 Windows 类型的程序设计技术，而下一章的操作更多地包含了 OLE 类型的程序设计技术。如果你想略去 OLE 控件的细节不看，你完全可以跳过下一章了。但如果你打算开发这些有用的控件，你一定要仔细研读下一章。

第十八章　用户方法和事件

我们先来看看我们将如何增强那些列表框控件的功能。

首先，那些控件是基于牢固的标准 Windows 列表框控件的。标准列表框是一种我们使用多年的工具了，并且它一直是作为控件中的控件或 VBX 中的控件被直接使用的。我们所要做的就是增加标签和列以及使用定制功能来显示图片或由 VB 程序员来显示图片。在我的 Windows 程序设计的班级中，这并不算什么难事，我的学生都要完成一个定制列表框的工程。但编写一个 OLE 控件，即使它基于牢固的子窗口类，你仍要面临大量纠缠不清的类、函数以及接口。所以需要我来引导你“穿过灌木丛”，在前面几章工作的基础上最终完成那三个控件编写。

前面几章我们为那三个控件实现了完全相同的一组属性。由此可见，它们看起来会很相似。但它们实际上是不一样的，现在我们就将它们区别开来。

18.1　TabList 控件的特殊性

TabList 控件是一种显示多列数据的列表框。并且列表框中每个文本表项都包含有一定数量的制表符。你知道，普通列表框将这些字符显示成小黑块。那是因为通常列表框是不具有 LBS_USETABSTOPS 风格的。我们的 TabList 控件则要设置这个风格：

```
BOOL CTabListCtrl::PreCreateWindow(CREATESTRUCT & cs)
{
        ⇓
        ⇓
cs.style|=LBS_USETABSTOPS;
return COleControl::PreCreateWindow(cs);
}
```

通常制表位(tab stop)是每隔32个对话框基本单位或8个字符宽设置一个。这可能并不符合你的要求。所以我们必须提供一个设置用户制表位的方法。

我们已经有了 Columns 属性。按通常规则实现的这个属性，可以用于 DrawList 和 ImageList 控件。但对于 TabList 控件来说，我们要将有关它的代码从 PreCreateWindow()中删除以便用作其它用途：

```
BOOL CTabListCtrl::PreCreateWindow(CREATESTRUCT & cs)
{
cs.ipszClass=T("LISTBOX");
//if(Columns>1)cs.style|=LBS_MULTICOLUMN;
        ⇓
        ⇓
cs.style|=LBS_USETABSTOPS;
return COleControl::PreCreateWindow(cs);
}
```

当然，Columns 的另一用途就是决定在我们新增的 TabStops 数组中有多少元素。(制表符的个数要比列数小1。)请注意区分 TabStops 这个用户属性和由容器管理的 TabStop 扩展属性之间的差异。

TabStops 属性是作为一个数组被加入到控件中的(见图18.1)。

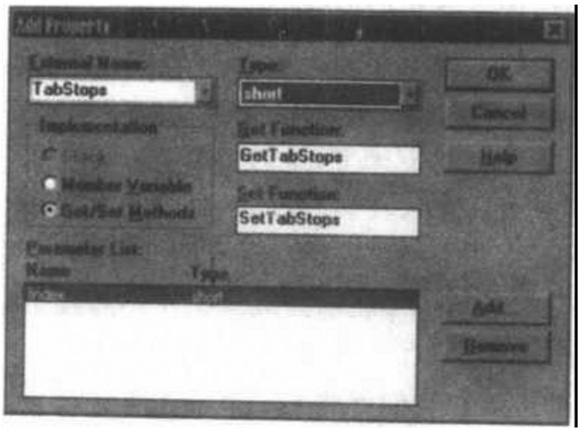

图18.1 增加 TabStops 属性

我们可以将这些制表位存贮在 MFC 数组类中。可用不着那么复杂。我无法想象在一个单列表框中的制表位会超过32个。所以我打算将这个属性存贮成一个简单的线性数组：

```
class CTabListCtrl: public COleControl
 {
          ⇓
          ⇓
private:
short NewIndex;
enum {MaxTabStops=32};
short TabStops [MaxTabStops];
          ⇓
          ⇓
 }
```

因此，Get 函数应按下面编写：

```
short CTabListCtrl:: GetTabStops(short Index)
 {
if(Index>=0&&Index<(Columns-1)) return TabStops [Index];
else
 {
  ThrowError(CTL_E_INVALIDPROPERTYARRAYINDEX,IDS_INVALIDINDEX);
  return 0;
   }
 }
```

但当用户设置这个属性值时，我们不仅要保存它，而且还要将它发送到列表框控件中去：

```
void CTabListCtrl:: SetTabStops(short Index,short New Value)
 {
```

```
if(Index>=0&&Index<(Columns-1))
   {
  TabStops [Index] =NewValue;
  ResetTabs();
    }
  else
          ThrowErro ( CTL_ E_ INVALIDPROPERTYARRAYINDEX, IDS_
INVALIDINDEX);
  SetModifiedFlag();
  }
```

我将发送 LB_SETTABSTOPS 消息的操作单独放到了一个函数中，因为当 Columns 改变时，也需要发送这个消息。

你一定还记得，Columns 属性在 DrawList 和 ImageList 控件中是以成员变量的形式实现的。但在 TabList 控件中我们需要将它实现成 Get/Set 类属性。这意味着要将它作为属性(short 型)加入到类的说明中。它的 Get 函数是非常简单的：

```
short CTabListCtrl:: GetColumns()
 {
  return Columns;
  }
```

但我们不能允许用户设置比33(它 TabStops 数组中元素总数大1)还要大的 Columns 值：

```
void CTabListCtrl:: SetColumns(short NewValue)
 {
  if(New Value>0&&New Value<=(MaxTabStops+1))
 {
Columns=New Value,
ResetTabs();
SetModifiedFlag();
  }
else
  ThrowError(CTL_E_INVALIDPROPERTYARRAYINDEX,IDS_INVALIDINDEX);
  }
```

ResetTabs()函数很有意思，因为它必须执行一个转换。列表框的制表位是按对话框基本单位来计算的，但 Visual Basic 开发者一般使用 twip 映射方式。但 Visual Basic 并不是唯一的 OLE 控件容器，其它开发环境可能会使用另外的坐标系统。此外，Windows API 函数 GetDialogBaseUnits()返回系统字体的按基本单位计算的大小。当然对话框中并不一定非要使用系统字体。

OLE CDK 提供了一些可利用的函数。算法并不算很复杂，它包含有二块代码：

```
void CTabListCtrl:: ResetTabs(void)
 {
  if(Columns>1)
```

```
{
CFontHolder FontHolder=InternalGetFont();
TEXTMETRIC TextMetrics,
GetFontTextMetrics(&TextMetrics,FontHolder);
WORD DlgWidthUnits=TextMetrics. tmAveCharWidth/4;
WORD DlgHeightUnits=TextMetrics、tmHeight/8;

int TabStopsX [MaxTabStops];
POINTL HiMerticPoint;
POINTF ContainerPoint;
ContainerPoint. y=o;
SIZE ListboxPoint;
CWindowDC dc(CWnd::FromHandle((HWND)GetHwnd()));
for(inti=0;i<MaxTabStops;i++)
    {
     ContainerPoint. x=TabStops [i];
     ContainerPoint. y=0;
     TransformCoords(&HiMetricPoint,&ContainerPoint,
                    XFORMCOORDS_ POSITION | XFORMCOORDS_ CONTAINER-
TOHIMETRIC);
     ListboxPoint. cx=(int)HiMetricPoint. x;
     ListboxPoint. cy=(int)HiMetricPoint. y;
     dc. HIMETRICtoDP(&Listboxpoint);
     TabStopsX [i] =ListboxPoint. cx;
      }
   SendMessage(LB_SETTABSTOPS,MaxTabStops,(long)(LPSTR)TabStops);
     }
  }
```

这个函数的主要操作是首先将制表位从 Container 坐标转换为 himetric 坐标；再将 himetric 坐标转换成列表框能处理的对话框坐标。最后将转换完毕的坐标发送到底层的列表框控件中去。

由于列表框控件可以自动处理多列显示，所以我们不需再增加属性了。如果要测试它，我们只需向列表框中增加内容就可以看到效果了。

在 Class Wizard 对话框的 OLE Automation 标签页中，我们可以向控件中增加属性也可以增加方法。其操作过程也是非常相似的。实际上，属性和方法只是在概念上的不同，而实现上是差不多的。最大的差异是容器所提供的存取它们的方式。大多数情况下，对属性赋值是使用赋值操作符，而方法则要使用函数调用。

既然这样，我们可以增加我们的第一个方法 AddItem。我们打算模仿 VBX 使用的 AddItem 方法，它允许带有可选的第二个参数，但 OLE 控件并不支持可选参数，并且也不允许所谓“重载方法”之类的东西。所以我们或者删掉第二个参数(它规定了插入位置的索引)；或

者编写两个不同名字的方法(比如,AddItem 和 InsertItem)。我愿意采用第二种方法。但我们现在只需要 AddItem。

当增加方法时，我们必须规定好参数的名字和类型。AddItem 需要一个参数：Item，它的类型是 BSTR。BSTR 不是一个类，它与标准 C 风格的字符串类似。所不同的是字符串前面存有字符串的长度。所以 BSTR 在许多时候可以被用作 C 风格的字符串。

Class Wizard 会为 AddItem 生成合适的发送(dispatch)以及下面我们要填写的函数：

```
void CTabListCtrl::AddItem(BSTR FAR* Item)
{
  SendMessage(LB_ADDSTRING,0,(LPARAM)(LPVOID)Item);
  SetModifiedFlag();
}
```

下面，为了模拟真正的列表框控件，我们打算加入对诸如单击(click)、双击(double click)等事件的支持。事件使 OLE 控件与 OLE 服务器相区分开来。同属性一样，事件也有库存事件和用户事件。加入库存事件是非常简单的，所以我将这个任务交给你自己来完成。最有意思的是用户事件。但我们还是从 ImageList 开始接触用户事件。

18.2 ImageList 控件的特殊性

ImageList 控件的特殊性在于它显示的不是文本，而是一组图片。OLE 提供了大量的各种各样的功能，可真正满足我们需要的太少了。

前面你已见过 CFontHolder 类的使用了。我并没有详细介绍它。但还有一个 CPictureHolder 类。这两个类有很多相同之处。那是因为它们的内部实现都与 OLE 中的各种类交织在一起，并且字体和图片都不是直接存取的，而是通过 OLE 接口来存取的。

在 ImageList 控件中，List 属性是一个图片的数组而不是文本的数组。这就是说，当使用 Class Wizard 向控件中加入 List 属性时，数据类型必须指定为 LPPICTUREDISP。

许多控件都有一个单图片属性;实际上是一个库存图片属性以及支持它的库存属性页。你也可以增加 Get/Set 类的用户图片属性。下面是 ImageList 属性页的宏：

```
BEGIN_PROPPAGEIDS(CImageListCtrl,4)
  PROPPAGEID(CImageListPropPage::guid)
  PROPPAGEID(CLSID_CColorPropPage)
  PROPPAGEID(CLSID_CFontPropPage)
  PROPPAGEID(CLSID_CPicturePropPage)
END_PROPPAGEIDS(CImageListCtrl)
```

同库存字体属性页一样，只要你的图片属性不是数组属性，图片属性页会自动装入它们。数组属性在设计时是不能被装入的。

如果你希望自己管理图片属性，你可以向控件类增加一个 CPictureHolder 数据成员。但我们需要的是一个 CPictureHolder 数组，并且 ImageList 控件又是基于列表框的，所以我们可以用它来为我们存贮数值。

定制列表框使用与常规列表框一样的 LB_ADDSTRING、LB_DELETESTRING 等消息。所不同的是，消息参数中提供的不再是字符串缓冲区，而是你给定的32位的数值。这个32位的

值可以是指向任何对象的指针。

使 ImageList 成为一个定制列表框，我们只需在函数中设置两个标志(LBS_OWNERDRAWFIXED 和 LBS_OWNERDRAWVARIABLE)中的一个就可以了：

```
BOOL CImageListCtrl:: PreCreateWindow(CREATESTRUCT & cs)
 {
  cs. lpszClass=T("LISTBOX");
  if(Columns>1)cs. style|=LBS_MULTICOLUMN;
  if(IIntegralHeight)cs. style|=LBS_NOINTEGRALHEIGHT;
  if(MultiSelect==1)cs. style|=LBS_MULTIPLESEL;
  else if(MultiSelect==2)cs. style|=LBS_EXTENDEDSEL;
  cs. style|=LBS_OWNERDRAWFIXED;
  return COleControl:: PreCreateWindow(cs);
  }
```

LBS_OWNERDRAWFIXED 风格规定了表项的高度是固定的；如果列表框中表项的大小是不同的，则要设置 LBS_OWNERDRAWVARIABLE 风格。

和 TabList 一样，ImageList 也有一个 AddItem 方法。但在 ImageList 中，这个方法带有一个 LPPICTUREDISP 参数。在这个方法中我们要生成一个 CPictureHolder 对象并把它传递到底层列表框中：

```
void CImagteListCtrl:: AddItem(LPPICTUREDISP Item)
 {
CPictureHolder* NewItem=new CPictureHolder;
NewItem->SetPictureDispatch(Item);
 NewIndex = (short) SendMessage (LB_ ADDSTRING, 0, (LPARAM) (LPVOID)
NewItem);
SetModifiedFlag();
}
```

我们不是在堆栈上生成 CPictureHolder 对象，而是使用 new 操作符。最初这个对象并不包含任何图片。你可以使用 CreateEmpty()方法来生成一个“空”图片。但我们想要用函数的参数生成图片。这需要调用 SetPictureDispatch()方法。调用这个方法后，我们就可以将 NewItem 指针通过 LB_ADDSTRING 消息传递给底层列表框了。

既然存在表项，就需要获取该表项的 Get 函数：

```
LPPICTUREDISP CImageListCtrl:: GetList(short Index)
 {
if(IsValidIndex(Index))
   {
   CPictureHolder* NewItem;
     SendMessage (LB_ GETTEXT, (WPARAM) Index, (LPARAM) (LPVOID)
&NewItem);
   return NewItem->GetPictureDispatch();
     }
```

```
  else
    {
        ThrowError (CTL_ E_ INVALIDPROPERTYVALUE, IDS_ INVALID-
PROPERTYVALUE);
      return NULL;
        }
  }
```

LB_GETTEXT 消息从列表框中返回一个32位的值并把它赋值给 CPictureHolder 的指针。而 GetPictureDispatch()方法会返回容器所需要的 LPPICTUREDISP 值。

你可能会奇怪，为什么我没使用 LB_GETITEMDATA 消息来存取那个值。对于定制列表框来说，LB_ADDSTRING 消息所提供的32位值会被复制到对应索引的表项数据中。但那只是一种拷贝。并且我们的 ImageList 控件和 TabList 类似，有一个 ItemData 数组属性，所以那个拷贝有可能被复盖。实际上，你可以在 List 属性的 Set 函数中看到这一点：

```
void CImageListCtrl:: SetList(short Index,LPPICTUREDISP NewValue)
 {
  if(IsValidIndex(Index))
     {
      DWORD ItemData=SendMessage(LB_GETITEMDATA,(WPARAM)Index);
      CPictureHolder* NewItem;
        SendMessage (LB_ GETTEXT, (WPARAM) Index, (LPARAM) (LPVOID)
&NewItem);
      delete NewItem;
      SendMessage(LB DELETESTRING,(WPARAM)Index);
      NewItem=new CPictureHolder;
      NewItem->SetPictureDispatch(NewValue);
      NewIndex=(short)SendMessage(LB_INSERTSTRING(WPARAM)Index,
                      (LPARAM)(LPVOID)NewItem);
      SendMessage(LB_SETITEMDATA,(WPARAM)NewIndex, ItemData);
      SetModifiedFlag();
        }
      else
      ThrowError(CTL_E_INVALIDPROPERTYVALUE,IDS_INVALIDPROPERTY-
VALUE);
  }
```

当用户想将一个图片替换成另一图片时，SetList()将被调用。列表框不支持这种替换操作。所以你必须删除要替换的表项并在该位置上插入新表项。但这样做会丢失原表项的表项数据值。所以，在上面的代码中我们必须首先保存该表项数据，交换完数据后再恢复它。

既然我们有了由列表框安全管理的数据，那么我们什么时候以及如何画出它们呢？

在第六章你已经看到了消息(如 WM_CTLCOLOR)是如何被 OLE 系统从底层控件被反射回 OLE 控件类中。定制列表框也生成应该被反射回来的消息。它们是当列表框具有排序风

格时，用来决定表项间顺序的 WM_COMPAREITEM 消息，决定表项显示所需要的空间大小的 WM_MEASUREITEM 消息、进行绘画的 WM_DRAWITEM。每种消息都带有一个指向具有特殊用途结构的指针，这个指针放在 lParam 参数中。

当列表框具有 LBS_OWNERDRAWEIXED 风格时，当它被生成时只向它的父窗口发送一次 WM_MEASUREITEM 消息。那是因为所有的表项都被认为具有相同的大小。但如果列表框具有 LBS_OWNERDRAWVARIABLE 风格，列表框将为每个表项都向父窗口发送一次 WM_MEASUREITEM 消息。

同 WM_CTLCOLOR 消息的处理一样，要处理这些消息必须经过两步：第一，为它们增加标准消息处理器；第二，在消息映射宏中增加"ocm"消息的映射：

```
BEGIN_MESSAGE_MAP(CImageListCtrl,COlelontrol)
  // { {AFX_MSG_MAP(CImageListCtr()
  ON_WM_CTLCOLOR()
  ON_WM_DRAWITEM()
  ON_WM_MEASUR EITEM()
  ON_MESSAGE(OCM_COMMAND,OnOcmCommand)
  //}} AFX_MSG_MAP
  ON_OLEVERB(AFX IDS VERB PROPERTIES,OnProperties)
  ON_MESSAGE(OCM_CTLCOLOR,OnComCtlColor)
  ON_MESSAGECOCM_DRAWI TEM_on ocmDraWItem
  ON_MESSAGE(OCM_MEASUKEITEM,OnOcmMeasureItem)
END_MESSAGE_MAP()
```

你必须编写这两个函数：OnOcmDrawItem()和 OnOcmMeasureItem()，并把它们的原型加入到类的说明中。这几个函数的主要工作实际上就是调用相应的 Windows 基本消息处理函数：

```
  LRESULT CImageListCtrl:: OnOcmMeasureItem ( WPARAM wParam, LPARAM lParam)
   {
  onMeasureItem((mt)wParam, (LPMEASUREITEMSTRUCT)lParam);
  retrun ();
    }
  LRESULT CImageListCtrl:: OnOcmDrawlItem(WPARAM wParam,LPARAM lParam)
   {
    OnDrawItem((int)wParam, (LPDRAWITEMSTRUCT)lParam);
    return ();
    }
```

你可能以为计算图片显示区的大小是件非常容易的事，但 CPictureHolder 类没有提供任何手段来获取它所包含图片的大小。大多数图片都有非常适合显示的大小，可我们无法找出它到底是多大。

所以我们不去测量图片的大小，而是在对 WM_MEASUREITEM 消息的响应中人为地给定一个大小：

```
void CImageListCtrl::OnMeasureItem(int nIDCtl,
        LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
lpMeasureItemStruct->itemWidth=80;
lpMeasureItemStruct->itemHeigth=60;
}
```

由于 WM_MEASUREITEM 消息需要的是设备坐标(象素点)，所以上述值可以在中等分辨率(比如800X600)的屏幕上横着可以显示10个图，竖着可以显示10个图。

对 WM_DRAWITEM 响应就是画图：

```
void CImageListCtrl::OnDrawItem(int nIDCtl,LPDRAWITEMSTRUCT lpDrawItem-
Struct)
{
if(int)lpDrawItemStruct->itemID>-1)
  {
    CDC*dc=CDC::FromHandle(lpDrawItemStruct->hDC);
    CPictureHolder*Item;
    SendMessage(LB_GETTEXT,(WPARAM)lpDrawItemStruct->itemID,
            (LPARAM)(LPVOID)&Item);
    RECT ImageRect;
    ImageRect. top=lpDrawItemStruct->rcItem. top;
    ImageRect. left=lpDrawItemStruct->rcItem. left;
    ImageRect. bottom=ImageRect. top+60;
    ImageRect. right=ImageRect. left+80;
    Item->Render(dc,&ImageRect,ipDrawItemStruct->rcItem);
  }
}
```

你一定要知道，即使列表框是空的，这些消息也会被发送。这就是为什么我们必须检测该表项是否有效的原因(空列表框的 itemID 为-1)。DRAWITEMSTRUCT 结构中其它标志会告诉你该表项是否处于被选择状态，是否具有输入焦点等等。大多数应用程序都对这几种状态做不同的绘图处理。

一旦我们获得了 CPictureHolder 表项，我们就可以简单地调用它的 Render()成员函数来画出表项。这比直接处理位图要省事多了！

可是当我们测试它时，我们发现它却不能正常工作。我编写了一个测试程序，使用下面代码在列表框中放入三个位图：

```
Private Sub Commu1andl_Click()
ImageList1. AddItem Image1. Picture
ImageList1. AddItem Image2. Picture
ImageListl. AddItem Image3. Picture
End Sub
```

运行的效果见图18.2。

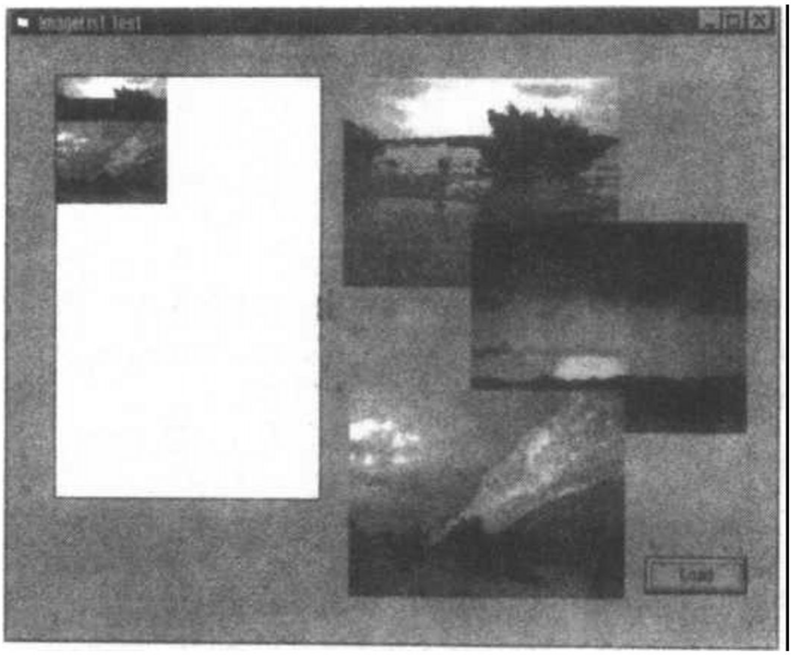

图18.2 运行不正常的 ImageList 控件

你看，ImageList 控件显示了三个图片，可这三个图片一个压着一个。实际上，每个图片是按照每行文本的高度来显示的，就好象 ImageList 是一个标准列表框一样。

进一步调试证明，当前 MFC 和 OLE 控件开发工具箱的版本中，“反射器”窗口并不反射 WM_MEASUREITEM 消息。显然，它们的 switch 语句中漏掉了该消息。所以我们定义的消息处理器永远也不会被调用，表项的高度和宽度也就不会被设置。

希望16位的 VisualC++其它版本会去掉这个错误。但我们实现基于定制列表框的 OLE 控件的任务不能完成了。当然你可以用另一种方法：将列表框作为控件的子窗口，而不是作为它的子类。这种方法固然非常费时费力，但却不会遇到这个麻烦。

18.3 实现用户事件

由于 WM_MEASUREITEM 对应的方法不能被调用，当然我们也同样无法完成 DrawList 控件。但它可以演示如何实现用户事件。它实现用户事件的方法是将 WM_COMPAREITEM、WM_DRAWITEM 和 WM_MEASUREITEM 消息转换成容器可以做出反应的事件。向工程中加入事件需要使用 Class Wizard 中的 OLE Event 标签页(见图18.3)。

要加入一个事件，需要点击 Add Event 按钮。同属性一样，这里没有“edit event(编辑事件)”按钮，所以你必须保证事件的正确性。否则你就必须删除错的事件，再重来一次。图18.4显示了 Add Event(增加事件)对话框。

并且同属性参数一样，你一定要事先规定好事件所带的参数以及它们的顺序。如果错了，你只有删除事件并重新开始。

一旦你增加了事件，你就会发现类的说明被修改了：

```
class CDrawListCtrl: public COleControl
{

        ⇓

        ⇓
```

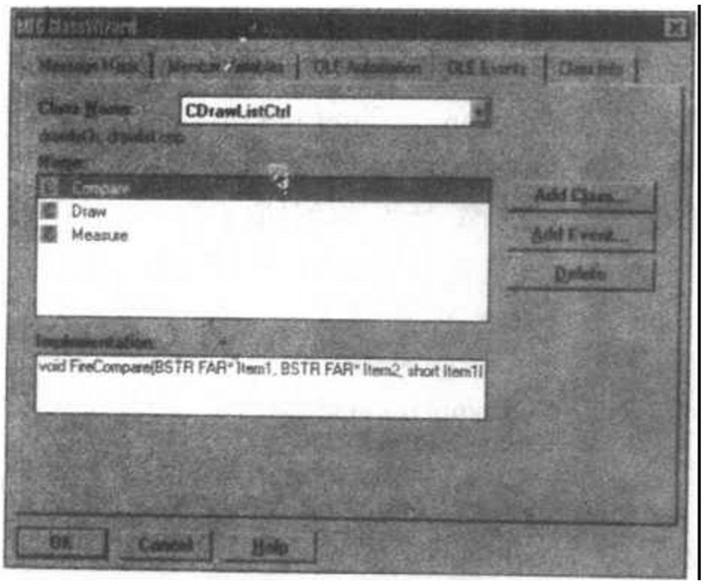

图18.3 Class Wizard 的 OLE Events 标签页

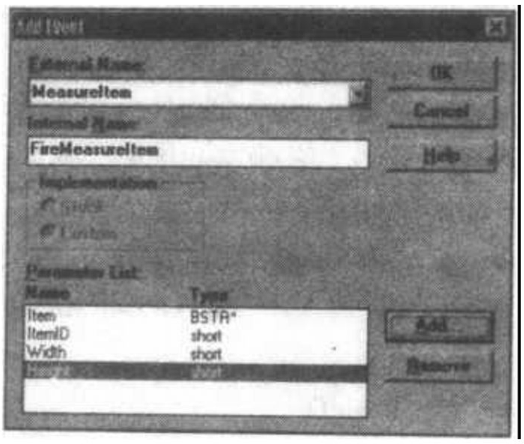

图18.4 增加事件对话框

```
//Event maps
  //{{AFX_EVENT(CDrawListCtrl)
  void FireDraw(OLE_HANDLEDC,BSTR FAR* Item,short x,short y,short Width,long
Height,
        short ItemIndex)
    {
  FireEvent(eventidDraw,EVENT_PARAM(VTS_HANDLE VTS_PBSTR VTS_I2
        VTS_I2   VTS_I4   VTS_I2), DC, Item, x, y, Width, Height,
ItemIndex);
    }
  void FireCompare(BSTR FAR* Iteml,BSTR FAR* Item2,short ItemlIndex,
        short Item2Index,short FAR* Comparison)
   {
  FireEvent(eventidCompare,EVENT_PARAM(VTS_PBSTR VTS_PBSTR VTS_I2
          VTS_I2 VTS_PI2), Iteml, Item2, ItemlIndex, Item2Index,
Comparison);
    }
  void FireMeasure (BSTR FAR* Item, short ItemIndex, short Far* Width, short FAR*
Height)
   {
  FireEvent(eventidMeasure,EVENT_PARAM(VTS_PBSTR VTS_I2 VTS_PI2
        VTS_PI2),Item,ItemIndex,Width,Height);
    }
  //}} AFX_EVENT
  DECLARE_EVENT_MAP()
        ⇓
        ⇓
```

```
};
```

这些事件都是内嵌函数，所以你不会在实现文件中找到它们的代码。其中真正起作用的函数是 FireEvent()，好在它的参数调用顺序被生成的 FireCompare()、FireDraw()以及 FireMeasure()函数隐藏了。

同 ImageList 一样，DrawList 实现了处理这些消息的函数。所不同的是它们的内部代码。ImageList 中的这些函数是作图的，而 DrawList 则将这些操作转交给了容器。比如，OnCompareItem 函数是这样定义的：

```
int CDrawListCtrl:: OnCompareItem(int nIDCtl,
                    LPCOMPAREITEMSTRUCT lpCompareItemStruct)
 {
  short Result;
  BSTR Templ=GetList(lpCompareItemStruct->itemID1);
  BSTR Temp2=GetList(lpCompareItemStruct->itemID2);
  FireCompare(&Temp1,
  &Temp2,lpCompareStrucl->itemID1,lpcompareIt emstruct-7item IP2.
           &Result);
 SysFreeString(Templ);
 SysFreeString(Temp2);
 return Result,
 }
```

虽然 DrawList 控件是一种定制的控件，但它的 List 数组属性包含的仍是字符串。(开发者可以利用这些字符串或数字的 ItemData 数组属性来决定画些什么。)GetList()函数返回一个 BSTR 字符串。但 FireCompare()函数的参数却是指向 BSTR 的指针。所以我们必须将 GetList()的返回值赋给暂时变量，以便将它们的地址传给 FireCompare()函数。FireCompare()函数只有当容器对 Compare 事件处理完毕后才能返回。

GetList()调用 CString:: AllocSysString()来返回一个 BSTR。当我们调用它时，我们必须调用 SysFreeString()以释放那些字符串。

OnDrawItem()和 OnMeasureItem()的定义也是类似的：

```
void CDrawListCtrl:: OnDrawItem (int nIDCtl, LPDRAWITEMSTRUCT lpDrawItem-
Struct)
 {
 BSTR Temp=GetList(lpDrawItemStruct->itemID);
 FireDraw((OLE_HANDLE)IpDrawItemStruct->hDC, &Temp,
        lpDrawItemStruct->rcItem. top,
        lpDrawItemStruct->rcItem. left,
        lpDrawItemStruct->rcItem. bottom-lpDrawItemStruct->rcItem. top,
        lpDrawItemStruct->rcItem. right-lpDrawItemStruct->rcItem. left,
        lpDrawItemStruct->ItemID);
 SysFreeString(Temp);
 }
```

```
void CDrawListCtrl::OnMeasureItem(int nIDCtl,
                LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
BSTR Temp=GetList(lpMeasureItemStruct->itemID);
short Width, Height;
FireMeasure(&Temp,lpMeasureItemStruct->itemID,&Width,&Height);
SysFreeString(Temp);
lpMeasureItemStruct->item Width=Width;
lpMeasureItemStruct->itemHeight=Height;
}
```

当然，在当前的 MFC 版本中，DrawList 的这个 OnMeasureItem()也同样不会被调用。

18.4 结束语

这是本书的最后一章，但仍似乎有意犹未尽的感觉。编写 OCX 并非一蹴而就的事。它需要你仔细专研本书所提供的代码和解释说明，并自己动手去编写。不要只看着我去编，你自己也要真正动手去编。有关 Control Wizard 以及 MFC 各类的详细信息参见 Microsoft 公司发行的 Visual C++软件包中相关文档。

好了!到这里我的任务完成了，可你的任务才开始。祝你好运!

Document generated by Anna's Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
"filename": "MTA5NDM3NDMuemlw",
"filename_decoded": "10943743.zip",
"filesize": 49864747,
"md5": "af28d1798b7acfe4cca7384ce6c3f901",
"header_md5": "3034e7f2d44431eb18627306b6d66522",
"sha1": "bf79df388aa8bc438a44a0b232236d059eb39e9e",
"sha256": "ac561320dea146494c3d53e54570e20b46ac6226e73c4d13d960ad74d25b2e39",
"crc32": 780863268,
"zip_password": "",
"uncompressed_size": 53973015,
"pdg_dir_name": "Windows 95\u2559\u251c\u2557\u00ba\u2510\u256a\u255d\u25a0\u2561\u2500\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e_10943743",
"pdg_main_pages_found": 353,
"pdg_main_pages_max": 353,
"total_pages": 366,
"total_pixels": 2328560254,
"pdf_generation_missing_pages": false
}